CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Scalable Framework for Continuous Query Evaluations over Multidimensional, Scientific Datasets

Cameron Tolooee, Matthew Malensek, and Sangmi Lee Pallickara[*]

Colorado State University, Fort Collins, CO, USA

SUMMARY

Efficient access to voluminous multidimensional datasets is essential for scientific applications, 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. We have designed and developed a distributed updatable cache that ensures the query output to contain the most recent data arrivals. We have developed a dormant cache framework to address strains on caching capacity due to intensive memory requirements. The data to be stored in the dormant cache is selected using the cached continuous query scheduling algorithm that we have designed and developed. This approach is evaluated in the context of Galileo, our distributed data storage framework. This paper includes an empirical evaluation performed on Amazons AWS cluster and a private cluster. Our performance benchmarks demonstrate the efficacy of our approach.
Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Continuous Query; Galileo; Query caching; Time series data

1. INTRODUCTION

In recent years, we have witnessed significant increase in the data collected from sensor networks, observational equipment, and social network applications. These data applications enable real-time monitoring and control, where decisions are based on analyzing voluminous, dynamic data. The data from these 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, 2] and recently in cluster-based storage systems [3, 4]. It has also been explored in real-time data mining systems [5, 8]. 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

*Correspondence to: Colorado State University, Fort Collins, CO, USA

[9]. 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 [10], 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 a window size. Therefore, in addition to the nearly real-time monitoring purpose, scientists can use this feature for more complex analysis over the integrated dataset. We evaluate our ideas in the context of our Galileo system [11, 12, 13, 14]. 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, they can be added to an existing physical data file or a new data file can be created for the dataset.

1.1. *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.
- Redundant download of results from continuous queries will increase computer network traffic; efficient data reduction techniques are required.

1.2. *Overview of Approach*

The approach described here is based on our distributed storage framework, Galileo [11, 12, 13, 14]. 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 [16] 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.

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, called the *cached continuous query coordinator*, is assigned to cache the continuous query output using an election algorithm based on the workload.

To reduce memory consumption, we have designed and developed a dormant cache swapping framework that orchestrates movements of query results between memory and disk. Results of less frequently retrieved continuous queries are migrated to disk and only actively updated portions of continuous query are maintained in memory. Based on the pre-specified retrieval schedule, the system prepares new results and determines portions of the query result that must be migrated to disk. To maximize availability of resource within a storage node, we re-arrange the queries' retrieval schedules to avoid conflicts. Finally, the capacity of a node to schedule multiple queries without conflicts is also factored in to the coordinator election process.

1.3. 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. This paper presents the dormant cache swapping framework that migrates caches from memory to disk until the next update interval to free memory for other queries or data processing. We have designed and developed cached continuous query scheduling algorithm that allows the framework to plan the task executions as close to their predetermined time as possible while avoiding conflicts. We make effective use of our cache by identifying queries that are likely to benefit from memory residency. We have discussed data sampling techniques used in this system to reduce the amount of query output data. This paper also presents the evaluation measurements of the overhead for maintaining distributed updatable cache and the effect of using dormant cache swapping. The evaluation was performed on a designated cluster and Amazon Web Services' Elastic Compute Cloud.

Collectively, these techniques form a novel approach for calculating recurring queries and a useful alternative to windowing operations in traditional stream processing systems is introduced. In comparison to similar continuous query frameworks, we contribute the prospect of holistic continuous query evaluation by seamlessly combining the processing of incoming stream data with data previously stored in the system. Though the implementation and evaluation of this work lies completely within the contexts of our distributed storage system, Galileo, the general methodology of our approach can be applied to other storage frameworks. Each component of the continuous query framework has analogous counterparts in other applications; allowing a similar framework to be constructed even with different system architectures.

Portions of this paper have been published at The International Conference on Cloud and Autonomic Computing (CAC 2014) [15]. We have since extended the paper considerably by scaling our approach with the dormant cache swapping architecture, cached continuous query scheduling algorithms, updating related works, data sampling algorithms and conducting additional performance evaluations. We have included new evaluations of the previously published work, as well as the new work, using Amazon Web Services' Elastic Compute Cloud.

1.4. Paper Organization

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

2. BACKGROUND: GALILEO SYSTEM OVERVIEW

Galileo is a distributed data storage system for voluminous multi-dimensional, geospatial, time-series datasets [11]. Galileo is designed to assimilate observational data, which arrives 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.

Galileo's topology is organized as a zero-hop distributed hash table. DHTs provide a decentralized, highly scalable overlay network that allows for insertions and retrievals similar to that of a hash table; e.g. put(key, value), and get(key). The class of zero-hop DHTs, such as Apache Cassandra [20] and Amazon Dynamo [21], 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.

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 that defines a geospatial bounding box for the data points. 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 a temperature of 28.9 C. 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.

3. 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. We have discussed our fault tolerant scheme and user interaction mechanism in a previous publication [15]. This section will discuss the architecture and algorithm used in the distributed updatable cache to handle these challenges.

3.1. 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 [10]. 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

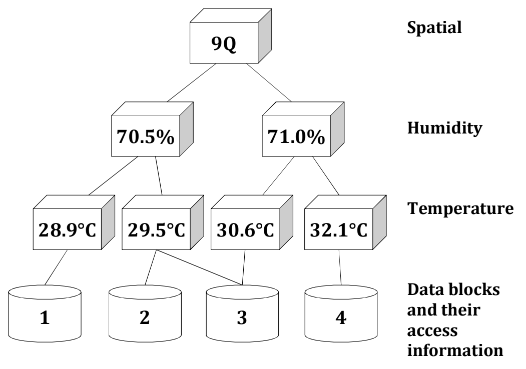


Figure 1. This is an example of a metadata graph for a dataset using three features to index data blocks: spatial geohash code, humidity, and temperature. Data block 4 contains data points that have 71% humidity with a temperature of 32.1 C for the geospatial area 9Q.

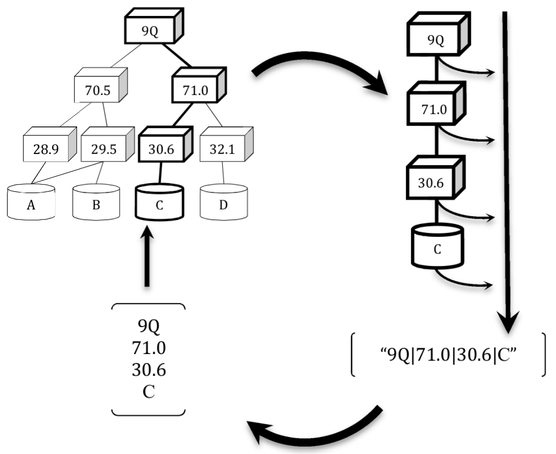


Figure 2. The compaction and expansion of a metadata graph path. The data compaction process provides a flattened representation of the metadata to maintain and track the dataset for continuous query effectively. Since it preserves the semantics of the metadata, it is easy to merge the sub-graph with the original query results when necessary.

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, 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. Queries with longer update intervals can efficiently use the dormant cache swapping framework, described in section V.

3.2. 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. 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 CCQs 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.

3.3. 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 their 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 results to the CCQ coordinator for compaction, also derived from the metadata graph.

3.4. *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 that 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 2. 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.

3.5. *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.

3.5.1. 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.

3.5.2. 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, CCQs provide an optional parameter that introduces compression into the caching process. When enabled, each label cache is streamed through a LZW compression [22] algorithm to reduce the volume of the cache. Due to the redundant nature of the cache label content, the high-performance LZW cache achieves excellent compression. Compression comes with a temporal cost and is not optimal in environments requiring concise response times. Thus, we made this an optional feature to best fit end users' needs.

3.5.3. 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. The dormant cache swapping system, discussed in section V, pushes this boundary even higher.

3.6. Election Algorithm

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 recently elected, will initiate processing cycles.

Figure 3 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.

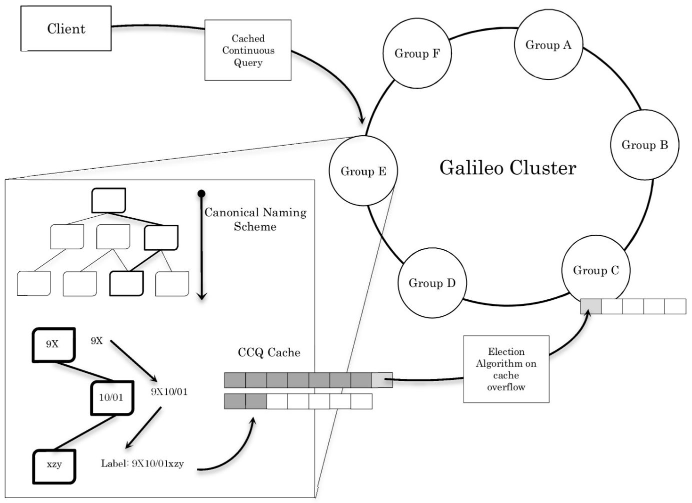


Figure 3. A high-level overview of the distributed updatable cache and cached continuous query progression. First, targeted nodes for the election phase are the nodes in the same group. If all of the nodes in the group are overloaded, the election process will be encompassed other groups based on the workloads and the network latency.

4. DORMANT CACHE SWAPPING FRAMEWORK

As mentioned previously, the distributed updatable cache has an upper limit in size directly related to the amount of cluster wide available memory. With that boundary in mind, the cached continuous query processing cycle displays a timely, repetitive pattern that can be leveraged to scale our solution to handle even larger queries. By writing dormant caches to disk, knowing they will not be needed until the next update interval, memory is freed for other queries or processing. As an interval for a cache on disk approaches, the contents are fetched in time for their update.

Consider the following scenario: a single storage node is responsible for maintaining two CCQs, A and B. Query A updates every five seconds and B updates every ten seconds. Both queries will be evaluated simultaneously every ten seconds. In the event that both query results are sufficiently large, they cannot both fit in memory at the same time and an election phase will be initiated to relocate one of the queries. This election can be avoided by offsetting the execution schedule of query B and writing the cache not being used in memory to disk. While this contrived scenario seems unlikely, even partial overlapping of queries can create this behavior. The technique of scheduling and swapping dormant queries ensures both queries have exclusive access to the main memory, avoiding the need to elect a node and migrate one of the CCQs.

A few challenges arise when taking this approach to scale this solution. To be able to efficiently implement this, we must ensure only one query cache is needed in memory during any phase of the evaluation. This solution also introduces a new metric to instantiating election phases since memory usage is no longer an accurate measure of a nodes work load.

4.1. Disk writes and prefetches

Each cache being maintained by the CCQ coordinator is only modified or updated during its processing cycle. The rest of its life is spent lying dormant, consuming valuable space in memory. Since each CCQ has a predefined interval at which it is to be evaluated, its executions can be scheduled in a way such that there are no two CCQs being evaluated concurrently on the same node. Doing so will guarantee only one cache be required in memory at any given time. This effectively allows every CCQ in the system to have complete access to memory resources without having conflicts with other CCQs.

By computing this schedule ahead of time, memory usage can be maximized by forcefully swapping out the dormant caches in exchange for the cache that is about to be updated in the next processing cycle. Each time a cache is finished updating, it is written to disk and the next CCQs cache will be read from disk into memory in time for its update.

4.2. Scheduling algorithm

The goal of the scheduling algorithm is to plan the task executions as close to their predetermined time as possible while avoiding conflicts. It is not always possible to have a perfect schedule where tasks are executed exactly on time without conflict. We choose to sacrifice accuracy in the interval resolution to guarantee no conflicts occur.

Each schedulable CCQ has 3 attributes: task ID, interval, and execution time. The task ID is a unique identifier for the CCQ, the interval is the user defined attribute that specifies the time in between query evaluations, and the execution time is the time taken to evaluate the query and swap the results to disk. The execution time is first calculated during the initial evaluation of the query and subsequently updated each time the query is executed. A schedule window is selected as the largest interval value of all the tasks. Choosing the largest interval as the scheduling window size, as opposed to the smallest interval or a fixed sized window, provides enough time to avoid conflicts while keeping the window size proportional to the CCQs being maintained. Window size minimums and maximums are in place to prevent extreme scenarios that exceedingly short or long windows can create. The tasks for the duration of the window are then scheduled from longest to shortest interval. In the case of a conflict, the CCQ with the longer interval is given priority and is preempted. Once a task is scheduled, its interval is maintained. If a task is scheduled late, the next interval for that task will be calculated from its late scheduled execution, rather than from the beginning on the schedule window. For example, if a query recurring every five seconds is scheduled two seconds late at time $t = 7$, the next scheduled execution will be at $t = 12$, (as opposed to $t = 10$).

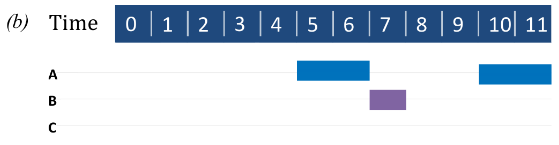
Because queries with longer intervals are given priority, checks must be in place to prevent the starvation of low interval queries. Starvation in this situation is a good indicator that the coordinating node is nearing its CCQ capacity. We use a few metrics to decide if an election phase is needed to prevent starvation. If even a single CCQ is unschedulable or the task window is filled beyond a threshold, one of the queries is relocated to another elected node. This technique ensures each node is being utilized to its maximum potential without creating latency during the evaluation.

Figure 6 demonstrates the algorithm by showing the first two schedule windows for the CCQs queue shown in the associated table. Window (b) has no conflicts and is scheduled exactly to the specifications of the queued CCQs. The second window, however, has a few conflicts, as shown in (c), that need to be resolved. Because task C has the longest interval value, it will be scheduled before its conflicting task, B. This will create a conflict between task B and task A at time 14. Task B has a longer interval and thus will be preempted. Window (d) show the completed second scheduling window. Once the schedule for the window is finished, the remaining time for each task is saved to be added on at the beginning of next scheduling window.

A few scenarios can arise that require modification of an executing schedule. When a new CCQ arrives to a node already in schedule, the currently executing schedule must be altered in light of the new CCQ. If a CCQ arrives during the execution of a schedule window, the remaining times for each of the CCQs are recorded, the schedule is ended prematurely, and a new schedule is created with the new CCQ included. The lightweight scheduling algorithm imposes less overhead than waiting for the schedule to finish. On the other hand, a query may expire in the middle of the window. Because the expiration is being tracked in each CCQ, it will be scheduled until its lifetime ends; choosing to complete an update interval even if doing so extends the lifetime slightly. This frees up slots in the schedule immediately allowing for a more open schedule.

(a)

Task ID	Interval	Execution time
A	5	2
B	7	1
C	12	3



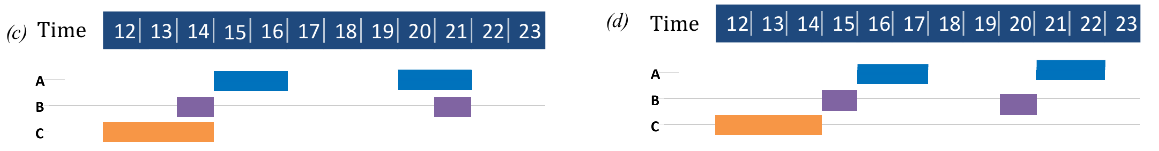


Figure 4. A demonstration of the dormant cache scheduling algorithm. (a) A table summarizing the attributes of the three tasks to be scheduled. (b) The schedule for tasks A, B, and C for the first schedule window. The windows size of 12 was selected from C's interval value which is the highest. No scheduling conflicts occur. (c) The second schedule window without conflict resolution. (d) The second schedule window with conflict resolution.

4.3. *Determining election phases*

Writing dormant caches to disk enables a more efficient use of the available memory, rendering the metric of memory usage as measure of work load less accurate. Using any measurement of node resources will reflect the usage of an individual query (not the overall nodes usage) because each query has exclusive access to those recourses. To combat this, changes in the technique of determining election phases must be made. Rather than looking at memory usage overall, election phases are determined by two factors: schedule usage and CCQ memory usage.

In the case that a CCQ is unschedulable without conflict or the schedule is filled beyond a threshold, this indicates that the node is no longer able to sufficiently maintain all of its cached queries. This scenario arises when a node is responsible for more queries than it can handle. In this situation, an election phase is executed to select a new node to manage one of the queries selected at random. Selecting the CCQ to be moved randomly, as opposed to selection based on a certain metric, ensures that a CCQ with a high metric does not get continuously moved around the cluster during periods of heavy loads. When a single CCQ grows too large to fit in memory an election occurs just as it did without the dormant cache exchange. A node will be selected according to the election algorithm, where only updates to the cache will be made.

4.4. Limitations

While the dormant cache swapping framework optimizes memory usage in many scenarios, there are suboptimal circumstances in which the framework provides little to no benefit. If a CCQ has a very rapid update interval, it will cause many scheduling conflicts. This will likely force election phases anytime there is another CCQ being maintained by the same node. The dormant cache swapping framework excels when query intervals are long and thus can be scheduled around one another. CCQs with short intervals will be forced to execute on a node without any competing CCQs and will perform as a distributed updatable cache would. The scheduling algorithm deals with this scenario well in that it will quickly move the problematic CCQ; however, it will not gain any of the benefits dormant cache swapping can provide.

5. SAMPLING QUERY RESULTS

Several classes of applications can benefit from sampling the records stored in Galileo to construct representative views of particular portions of the dataset. For instance, real-time data visualization using CCQ may require low-latency access to a broad range of information, but the particular data points in question do not necessarily have to be high resolution or inspected exhaustively to create the desired imagery. Samples are frequently used to generate summaries that provide high-level sketches of the data, which become increasingly important as datasets grow to sizes that make manual analysis untenable. These summaries can be used to glean insights from the data or to filter out unwanted records as a pre-processing step. We support this functionality in our framework through user defined sampling functions (UDSFs) that can be applied to a variety of sources: query results, cached records, metadata, and on-disk file blocks.

Because sampling is often domain specific, we provide user defined sampling functions (UDSFs), which allow users to define their own personalized sampling algorithms to be executed in parallel across the cluster. We provide a software interface that 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. UDSFs can also be stored temporarily on relevant nodes and reused in future queries.

We supply a set of predefined query algorithms that can be used directly or extended by users. These include random sampling, uniform sampling, and stratified sampling. Random sampling selects data blocks at random from the source provided, and can perform the sampling operation with or without replacement (where samples are returned to the pool, ensuring they are independent of one another). Uniform samples, on the other hand, take the distribution of values into account for each feature type and will produce output subsets that closely reflect the actual distribution of the data points. As records are streamed into the system, each feature distribution is updated in an online manner to facilitate this type of sample and will be maintained in memory as a probability density function (PDF). Finally, stratified sampling extends uniform sampling by decomposing the input distribution into disparate strata that are then sampled from individually. This process ensures a fair distribution of samples across the feature space while also accounting for dispersion; in essence, it allows for small portions of the dataset to still be represented in the final subset, a feature that is critical in applications such as anomaly detection.

In the context of the cached continuous query framework the user supplies their custom sampling algorithm or one of the algorithms provided by the framework and 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.

Sampling can be performed across a variety of sources to give users fine-grained control over the inherent timeliness-accuracy tradeoff of this functionality. We allow samples across query results and cached metadata paths in the CCQ environment, as well as from the metadata graph and on-disk file blocks. These sources represent an accuracy-timeliness gradient; records in the CCQ environment will be fast and highly accurate with a relatively narrow scope, while the metadata graph provides a broad but less precise view of the data. Finally, on-disk blocks represent the canonical representation of the data but require latency-intensive operations to retrieve. Interactive visualizations may sample from the results of a continuous query, whereas weekly reports could be generated by sampling from the metadata graph. In scenarios where complete accuracy is paramount, the samples will be taken from blocks stored on disk.

6. PERFORMANCE EVALUATION

To benchmark the effectiveness of our continuous query framework, testing was performed on the distributed updatable cache and dormant cache swapping separately. A set of benchmarks were evaluated in two environments. The first environment is a cluster dedicated to running the Galileo storage system for this data and the second on Amazon Web Services' (AWS) Elastic Compute Cloud (EC2) cluster in conjunction with AWS Elastic Block Store (EBS) for persistent storage. We sourced real-world data from the North American Mesoscale Forecast System (NAM) [23], which is maintained by the National Oceanic and Atmospheric Administration (NOAA). The NAM is run four times daily, 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).

6.1. Dedicated Cluster Evaluation

The first sets of tests were performed on a personal cluster dedicated to running Galileo. 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).

6.1.1. Distributed Updatable Cache We composed a series of test scenarios to measure three main aspects of our design: speed, memory consumption, and continuous retrieval throughput. To ensure the continuous cached queries perform as expected, we ran a series of retrieval throughput benchmarks in comparison with Galileo's standard query evaluation method. Figure 5 compares the retrieval times of our distributed updatable cache system versus a standard Galileo query and the dormant cache swapping query retrieval time to be discussed in subsection 2. By altering the query parameters, we adjusted the number of query results to range from 0 to 10,000. These results demonstrate that the time costs of compression, caching, and reassembly are less than that of the standard method: distributing queries to nodes in parallel, and then aggregating results. 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 to 1,000,000 file blocks to increase the relative severity of the memory increases we will incur. This benchmark involved issuing 10 distinct CCQs yielding 100,000 results each; effectively 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 CCQs were dispatched and the memory usage was measured once again, after the CCQs were able to equilibrate. Figure 6 represents the percentage of change in memory usage before and after the CCQs 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.

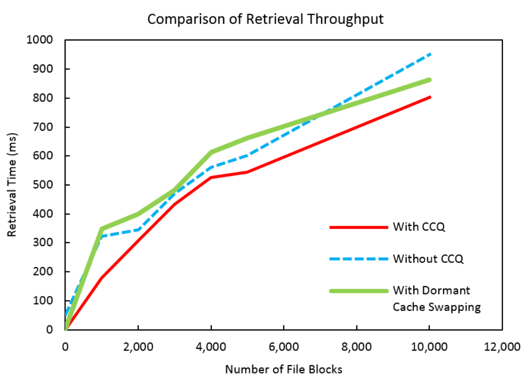


Figure 5. Comparison of retrieval performance of queries of varying sizes between standard and continuous cached queries.

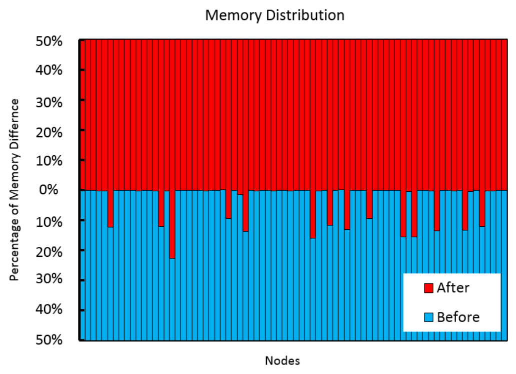


Figure 6. Difference in memory usage after deploying 10 continuous cached queries having 100,000 results each over a dataset with 1,000,000 file blocks.

Figure 6 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 6, 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 881 kilobytes.

Our final dedicated benchmark targets the continuous retrieval process. We deployed a CCQ into an empty cluster, with 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 I. Our batch processing technique allows for consistently fast retrieval times in the face of large query output volumes. Longer update interval times produce a larger number of file blocks retrieved. This is because there is more time for incoming data blocks to accumulate before a processing cycle is initiated. 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.

Table I. 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

6.1.2. Dormant Cache Swap The primary goal of the dormant cache swapping framework is to scale the distributed updatable cache solution while minimizing the negative impact to its performance. The same retrieval throughput experiments were performed with the dormant cache swap system. Figure 7 shows the results on the same plot as the other retrieval throughput benchmarks for comparison. While we observe degradation in performance for smaller query results, the retrieval time improves as the number of file blocks returned grows. Retrieving results from disk is a penalty that is costly with respect to small numbers of results. As the volume of the query results grow large, the overhead of evaluating the query in its entirety is greater than having to read the results from disk. This is observed at 10,000 file blocks where the dormant cache execution time is faster than the standard Galileo query. Because the dormant

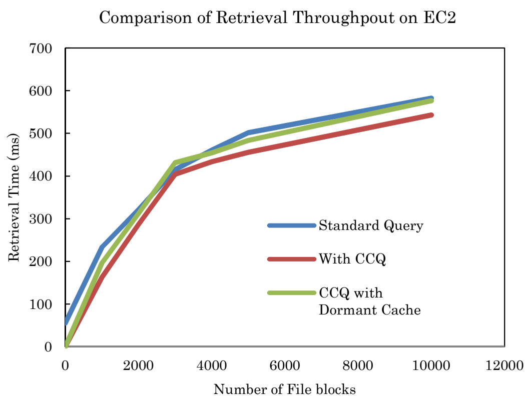


Figure 7. Comparison of retrieval performance of queries of varying sizes between standard and continuous cached queries within the EC2 cluster.

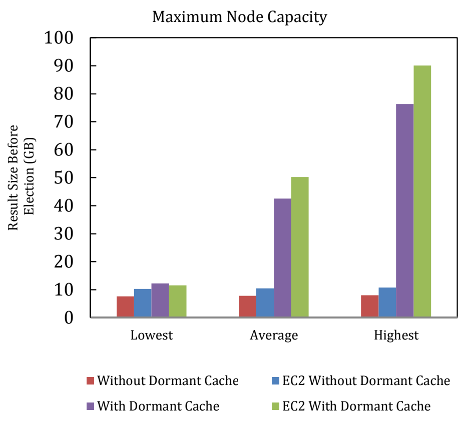


Figure 8. Storage node capacity of a node with and without the use of dormant cache swapping in both the dedicated cluster and the EC2 cluster.

cache is built on top of the distributed updatable cache framework, it will not outperform the retrieval throughput of the updatable cache until its bottleneck volume is reached.

Testing the memory usage of each computer in the cluster is not an accurate measure of the memory improvements. Because the caches in memory are constantly being swapped to and from disk, the usage is dependent on the size of the query currently being updated, not the node's query result capacity. The memory usage on a single node may be different at another time. Because of this, executing a test like that in Figure 6 will be inaccurate. Instead, we tested the capacity of a single node with and without the use the dormant cache swapping framework to demonstrate the scalability.

For this benchmark we selected a single HP DL160 server from our heterogeneous cluster to act as the node to bottleneck. The experiment was run as follows: a large set of continuous queries was generated with parameters being randomly selected from a group of predefined. This creates a query set having a wide range of performance altering parameters. The update interval is chosen to be a value between 5 and 300 seconds. Dynamic update intervals test the scheduling algorithm to establish non-trivial schedules, compared to a static interval. An infinite query execution time is selected to ensure that the cached queries do not expire before the experiment is complete. To vary the size of the generated queries, a 10% humidity attribute range selected to limit the volume of results.

One by one, each query is sent to the designated node without dormant cache swapping until an election phase is initiated due to a capacity limit. The total volume of query results is recorded, the node is restarted, and the process is repeated with dormant cache swapping enabled using the same query set. A total of ten query sets were generated to simulate many different scenarios. Figure 8 summarizes the results with the minimum, average, and maximum query result sizes for the experiment. In the tests without the use of the dormant cache swapping, there is little difference between the lowest, highest, and average volume of the results; which is to be expected. Because a node was singled out, it was pushed to its memory capacity before issuing an election phase. The tests with the dormant cache swapping framework display a very large range of improvement. The lowest maximum query result volume occurred in the query set that had many small intervals. Because of the framework's limitations with rapidly updating intervals, we see relatively small improvements.

In the average and highest plots, a very large discrepancy between the test cases is observed. The selected node was able to maintain many more queries in the test sets with longer update intervals. The maximum result size was realized with a query set containing mostly large intervals, allowing the node to maintain over nine times the amount of data than the best case without dormant cache swapping.

6.2. Amazon EC2 Cluster Evaluation

Amazon Web Services' (AWS) Elastic Compute Cloud (EC2) provides a cluster of on-demand virtual machines to run distributed computations. Users can launch as many virtual servers as needed and scale their cluster in either direction to fit their needs. Because EC2 removes the hardware requirement for warehouse-scale computing, its usage in many scientific applications, such as genomics, is expanding

[24]. EC2 used in tangent AWS storage capabilities, for instance Elastic Block Store (EBS), satisfy the requirements to benchmark our framework in a virtual environment.

To configure and manage the AWS EC2 cluster, we used StarCluster [25], a cluster-computing toolkit for EC2. This tool simplifies task of configuring and deploying cluster by automatically setting up hostnames, passwordless SSH, and NFS according to a specified configuration. Using StarCluster, we created a cluster consisting of 20 m3.xlarge instances. An m3.xlarge instance has the follow specifications: High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors, 15GB RAM, 80 GB SSD-based instance storage, 100 GB EBS General Purpose SSD volume, and a balance of compute, memory, and network resources. In the interest of comparing the performance of our continuous cached query framework in different a environment, we repeated many of the same benchmarks from the previous section in the EC2 setting. The same data set used previously was sourced for these experiments, though only the metadata was uploaded and the file blocks were discarded.

The first benchmark followed the exact same procedure as the previous CCQ throughput benchmark. Figure 7 shows the throughput of the different queries. To our surprise, the 20-node EC2 cluster outperformed the 77-node dedicated cluster in this benchmark. To investigate the explanation, we ran a series of network throughput experiments on in both environments to determine if it was the cause of the discrepancy. The results confirmed our suspicions in that the EC2 network benchmarks performed about 10% faster than the dedicated cluster. The dormant cache swapping throughput also realized a throughput improvement over the dedicated cluster, especially with small numbers of results. Because the instance's storage drives are all solid state with considerably faster disk IO than the dedicated cluster's hard disks, the overhead for the dormant cache swapping is lessened. Overall, the same trends between the standard Galileo query, cached continuous query without dormant cache swapping, and cached continuous query with dormant cache swapping have recurred. Examining the limits of a virtual server particularly piqued our interest. We were curious to see the performance implications of potentially competing with other virtual machines for resources. The same node capacity benchmark was conducted on an EC2 instance. As shown in Figure 8, the results indicate that the dormant cache swapping thrives in both environments. The direct comparison between the EC2 and dedicated clusters are not valid due to the difference in specifications; EC2 has more RAM, solid state drives, etc. However, the expansion of capacity the dormant cache swapping can provide is evident.

7. RELATED WORK

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

Initially, data streaming processing systems were exclusively used in sensor networks as the primary method of data analysis. Systems such as NiagaraCQ [3] and Borealis [4] innovated streaming processing techniques such as partial updating, in-place modification, and sliding window operators. Streaming and real-time processing systems have since evolved to accommodate the many data streaming applications found today. The Web 2.0 movement has pushed data streaming into functions in the direction of social network analytics and large text data-mining. Additionally, advances in sensor technology have dramatically expanded the use of sensor networks beyond scientific applications to commercial uses as well.

Apache Storm [5] is a centralized, hierarchical stream processing system on top of Nimbus [6]. Users submit a stream processing topology in the form of a graph. Each vertex specifies a spout or bolt representing a component in the stream processing pipeline. Nimbus distributes and coordinates the execution of the submitted topology to the compute cluster. Data can subsequently be streamed to a storage system after analysis in an independent manner. Because of this, only the incoming data can be used in the processing. Ad hoc and stateful processing becomes a challenge in that data that has already passed through the system cannot be processed or has to be re-streamed through the framework.

Apache Spark [7], a cluster computing framework initially developed at UC Berkeley, has integrated streaming analytics into its batch analysis infrastructure. Much like Galileo's continuous query system, Spark Streaming [8] accrues streaming data into small batches and processes each batch over its distributed computing infrastructure. Their concept of resilient distributed datasets (RDD) [17] act as the basic unit of storage. By distributing the RDDs across the cluster, large in-memory distributed computations can be conducted in an efficient, fault-tolerant fashion. This concept draws parallels between our updatable distributed caches. Compared to other methods on stream processing, like that of Storm or sliding window approaches, batch processing suffers a slight latency in performance. No live computations are being done for the duration that data is being accumulated thus adding a lag time in live results proportional to that duration.

The distributed updatable cache framework follows Hausenblas and Bijnens' lambda architecture system paradigm [18] of bridging the gap between data at rest and data in motion. The lambda architecture defines three layers within a distributed storage and computation framework: batch layer, speed layer, and serving layer. Data entering the system is streamed to both the permanent append-only database, the batch layer, as well as a data processing stream engine, the speed layer. The batch layer serves as a master store for all data in the system. Static views are generated in batches to create queryable snapshots of the data stored. Conversely, the speed layer processes the relatively small amounts of live data in an online fashion. Finally, the serving layer merges these two views of the incoming data based on user queries. Galileo's continuous query framework embodies ideas from the lambda architecture in that both streaming and stored data are eloquently combined to achieve holistic, old and new, timely evaluations.

Amazon Kinesis [19] 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. Short temporal continuous queries, such as the popular "calculate the top-K Tweets every n minutes," can easily be handled by Kinesis, but continuous queries spanning a large time frame are not feasible. For example, a query such as "How does the current weather compare with the weather on the same day last year? ...two years ago? ...in London? etc..." requires the processing of current data in tangent with old data already stored.

8. CONCLUSIONS AND FUTURE WORK

8.1. 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 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 cached continuous query coordinator at each node is responsible for initiating the query at the defined intervals. The schedulable nature of the CCQ intervals enables us to write caches lying dormant in memory to disk. This grants the query currently being evaluated full access to memory without competition.

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, the throughput of our distributed cached continuous queries, and the scalability of our framework with the dormant cache swapping system. Our benchmarks show the efficacy of our approach in multiple life-like environments of both a personal dedicated cluster and a cloud computing infrastructure. The system addresses the various challenges of maintaining continuous queries at scale. The batch processing techniques maintain the robust querying capabilities of Galileo while allowing for only updates since the last batch to be cached. The system eliminates the need for redundant querying, dramatically reducing network traffic.

8.2. Future Work

Our vision of holistic continuous query evaluation has many ideas for improvement and expansion. 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 sliding windows, over the incoming data. Such an implementation gives clients the choice in tradeoffs involved in the various continuous query techniques. Having to understand which querying techniques are optimal for the client's problem is a burden in and of itself. Autonomously determining which style of query will perform optimally is another future goal we are striving for.

ACKNOWLEDGEMENT

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018) and Amazon AWS in Education Research Grant Award (EDU_Research_Pallickara_ColoradoStateU_Spring2013).

REFERENCES

1. Y. Chen, K. Lwin and S. Williams, Continuous Query Processing and Dissemination
2. S. Kim, S. H. Son, and J. A. Stankovic, Performance Evaluation on a Real-Time Database, in Proc. IEEE Real-Time Technology and Applications Symposium, 2002.
3. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang: NiagaraCQ: a scalable continuous query system for Internet databases, in Proc. ACM SIGMOD international conference on Management of data, pp. 379-390, 2000.
4. D. J. Abadi et al., The design of the borealis stream processing engine, in Proc. Conference on Innovative Data Systems Research, vol. 5, pp. 277-289, 2005.
5. A. Toshniwal et al., Storm@twitter, in Proc. ACM SIGMOD International Conference on Management of Data, 2014.
6. K. Keahey et al., Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid, Scientific Programming Journal, vol 13, No. 4, Special Issue: Dynamic Grids and Worldwide Computing, pp. 265-276. 2005.
7. M. Zaharia et al., Spark: cluster computing with working sets, in Proc. 2nd USENIX conference on Hot topics in cloud computing, pp. 10-10. 2010.
8. M. Zaharia et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters, in Proc. *4th USENIX conference on Hot Topics in Cloud Computing*, USENIX Association, 2012.
9. Google Developers. (2013) Mobile Backend Starter. [Online]. Available: https://developers.google.com/cloud/samples/mbs/
10. B. Babcock et al., Models and issues in data stream systems, in Proc. ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 1-16, 2002.
11. M. Malensek, S. Pallickara, and S. L. Pallickara, "Expressive Query Support for Multidimensional Data in Distributed Hash Tables, in Proc. *IEEE/ACM Conference on Utility and Cloud Computing*, pp 31-38, 2012.
12. M. Malensek, S. Pallickara, and S. L. Pallickara, "Galileo: A Framework for Distrubuted Storage of High-Throughput Data Streams, in Proc. *IEEE/ACM Conference on Utility and Cloud Computing*, pp. 17-24, 2011.
13. M. Malensek, S. Pallickara, and S. L. Pallickara, "Exploiting Geospatial and Chronological Characteristics in Data Streams to Enable Efficient Storage and Retrievals, *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1049-1061, 2013.
14. M. Malensek, S. Pallickara, and S. L. Pallickara, "Enabling Access to Time-Series, Geospatial Data for On Demand Visualization, in Proc. *IEEE Symposium on Large-Scale Data Analysis and Visualization*, pp. 141-142, 2011.
15. C. Tolooee, M. Malensek, and S. L. Pallickara, "A Framework for Managing Continuous Query Evaluations over Voluminous, Multidimensional Datasets," in Proc. *IEEE Cloud and Autonomic Computing Conference*, London, UK. 2014.
16. G. Niemeyer. (2013) Geohash Web Service. [Online]. Available: http://geohash.org
17. M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in Proc. 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
18. M. Hausenblas and N. Bijnens. (2014) Lambda Architecture. [Online]. Available: http://lambda-architecture.net/
19. Amazon web services. (2013) Amazon kinesis Developer Guide. [Online]. Available: http://awsdocs.s3.amazonaws.com/kinesis/latest/ kinesis-dg.pdf
20. A. Lakshman et al., Cassandra: a decentralized structured storage system, ACM SIGOPS Op. Sys. Rev., vol. 44, no. 2, pp. 3540, 2010.
21. D. Hastorun et al., Dynamo: amazon's highly available key-value store, in symposium on Operating systems principles, pp. 205-220, 2007.
22. T. Welsh, A Technique for High-Performance Data Compression, Computer, vol. 17, no. 6, pp. 8-19, 1984.
23. NOAA. (2013) The NAM. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM
24. K. Konstantinos, et al., Cloud BioLinux: pre-configured and on-demand bioinformatics computing for the genomics community, BMC bioinformatics vol. 13, no. 1, pp. 42, 2012
25. The Software Tools for Academics and Researchers. (2013) StarCluster [Online]. Available: http://star.mit.edu/cluster/