Autonomous Data Management and Federation to Support High-throughput Query Evaluations over Voluminous Datasets

Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara, *Members, IEEE*

Abstract—In recent years, both the breadth and depth of information being generated and stored has continued to grow rapidly, causing an information explosion. Observational devices and remote sensing equipment are no exception to this rule, giving researchers new avenues for detecting and predicting phenomena at a global scale. To cope with these storage loads, hybrid clouds offer an elastic solution that also satisfies processing and budgetary needs.

This paper describes our algorithms and system design for dealing with voluminous datasets in a hybrid cloud setting. Our distributed storage framework autonomously tunes in-memory data structures and query parameters to ensure efficient retrievals and minimize resource consumption. To circumvent processing hotspots, we predict changes in incoming traffic and federate our query resolution structures to the public cloud for processing. We also demonstrate the efficacy of our framework on a real-world, petabyte dataset consisting of over 20 billion files.

Index Terms—E.1.C Distributed file systems, E.1.B Distributed data structures, C.1.4.A Distributed architectures

1 Introduction

Over the past decade, there has been an exponential growth in the amount of data that must be stored and managed. IDC estimates that in 2011, 1.8 zettabytes (ZB) of data was generated. One of the key contributors to this increase is networked observational devices; while the number of devices continues to rise, the resolution and frequency of their measurements have also increased. This leads to the accumulation of voluminous datasets that must be processed to extract knowledge.

Due to the data volumes involved, targeting the entirety of these datasets for ad hoc analysis is infeasible. Common types of analysis include hypothesis testing, construction of conditional probability tables in support for Hidden Markov Models, and creation of regression and ensemble models to make forecasts. A precursor to such analysis is interactive query instrumentation and evaluation with MapReduce that identifies portions of the dataset that are suitable for processing. Each observation may be multidimensional, with multiple features of interest. Furthermore, features may be derived from existing features. In these applications, queries play a significant role in processing and extracting knowledge by identifying data blocks (and storage nodes) over which computations should be performed. In our solution, results from queries are returned as graphs that include the metadata of matching results; leaves in these graphs contain locations of data blocks on disk. Our MapReduce computations consume these graphs as input and then processing activities are pushed onto relevant storage nodes to ensure data locality.

- *M. Malensek, S. Pallickara, and S. Pallickara are with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: {malensek,sangmi,shrideep}@cs.colostate.edu*

Given that our dataset is voluminous and new observations arrive at rapid rates, human intervention during any stage of the data management process is infeasible. Furthermore, query evaluations over the data must avoid disk accesses; since seek times are on the order of a few milliseconds, the performance overheads resulting from disk accesses would be immense. Rather, metadata must be extracted and maintained in memory-resident data structures as observations arrive. These memory-resident data structures are then used during query evaluations and instrumentation. The accuracy, timeliness, and volume of results depend on in-memory data structures that assist query evaluations.

1.1 Research Challenges

Supporting timely, accurate, high-throughput query evaluations and ad hoc analysis at scale over voluminous time-series datasets poses several unique challenges:

- ***Data volumes:*** The rates at which new observations are assimilated are high. We consider petascale datasets with billions of files.
- ***Autonomous data management:*** Given the data volumes and dimensionality of the feature space, human intervention is infeasible.
- ***Memory management:*** Data structures must be managed autonomously to strike a balance between memory consumption, resolution, and correctness.
- ***Query instrumentation:*** Interactive query instrumentation at scale poses several challenges, as the number of features and combinations of feature values that can be explored are quite high.

1.2 Research Questions

We considered several research questions in this study:

1) What options allow us to improve query instrumentation? It is often infeasible to manually search and identify the most relevant feature combinations.
2) How can we autonomously increase query evaluation efficiency? This involves managing query loads and ensuring efficient memory use.
3) How can we federate with public clouds to cope with increased demand while also avoiding expensive data hosting and movement costs?
4) When and how can we assimilate federated VMs under high load in a proactive manner that also accounts for trends and seasonality in access patterns?
5) How do we maintain consistency in a federated setting? Lightweight gossip protocols must be used to synchronize between the private and public clouds.

1.3 Overview of Approach

Our approach targets autonomously-tuned data structures, query evaluations, and load balancing operations in a hybrid cloud setting that includes combinations of both private and public clouds. We provide *fuzzy queries* with a SQL-like syntax that allow users to locate records that are most relevant to their query parameters, sidestepping the traditional iterative query refinement process. To avoid unnecessary processing, we autonomously resize bloom filters that are used to determine whether or not query inputs will produce results. In situations where high loads overwhelm nodes in the cluster, we federate processing activities to the public cloud by predicting incoming load changes and launching VMs to compensate. In a federated setting, index structures are optimized to improve memory usage and reduce turnaround times. The algorithms and designs discussed in this work were implemented in the context of our multidimensional distributed storage framework, Galileo, but are broadly applicable to other systems.

1.4 Paper Contributions

This study describes our framework for autonomous data management at scale using federated clouds. Specific contributions include:

- We demonstrate how to autonomously tune data structures and query evaluations as the feature space evolves. Our approach is able to reduce memory footprints without compromising on accuracy.
- We have identified aspects of voluminous data management that are amenable to federation. Our framework federates in situations where query instrumentation and evaluation must be performed efficiently.
- We have incorporated support for both proactive and reactive cloud bursting strategies. Our proactive scheme accounts for trends and seasonality in the queries received at individual storage nodes.
- Our framework addresses the issue of consistency in federated settings. We rely on eventual consistency, but are able to prioritize state updates between the private cluster and public cloud based on their impact and importance to query evaluations.

1.5 Related Work

While our cloud bursting approach focuses on moving computational loads associated with query resolution to the public cloud, several storage frameworks support migration of physical data in response to increased load; Cloudy [1] is a multi-paradigm storage system based on Cassandra [2] that implements a reactive cloud bursting strategy orchestrated by its *Cloudburster* component. Similarly, Bicer et al. [3] describes a system for data-intensive computing that incorporates cloud bursting for storage scalability.

MongoDB [4], HDFS [5], and HBase [6] all support expanding and contracting their respective resource pools, which may include VMs in the cloud. Each system is designed around specific data models to best serve their problem domains. However, a distinction is not made in these systems between nodes in the private and public clouds.

1.6 Experimental Dataset and Test Environment

The test dataset used in this study was collected from the NOAA North American Mesoscale Forecast System (NAM). The NAM contains atmospheric feature data including spatiotemporal attributes, surface temperature, precipitation, cloud cover, visibility, and pressure. We sampled from the NAM to create our dataset of 20 billion files. Each file contained 10 KB of raw feature data and 40 KB of graphical tiles for a total dataset size of 1 petabyte.

Our benchmarks were carried out on a heterogeneous private cloud consisting of 312 VMs. Each VM was allocated a single processor core, 2-4 GB of RAM, access to a shared gigabit network link, and a physical hard disk for storage. Host machines ran Fedora 20 and the KVM hypervisor, while guest VMs ran Fedora 21. Our public cloud VMs ran Red Hat Enterprise Linux 7.1 on Amazon Elastic Compute Cloud (EC2) under Xen HVM. In an effort to minimize the physical distance from our private cloud in Colorado, USA, EC2 instances were launched in the `us-west-1` regions. Galileo was run under OpenJDK version 1.7.0_75, and we used EC2 API Tools 1.7.3.2.

2 System Overview

Our distributed storage framework, Galileo, was designed for high-throughput management of multidimensional data [7], [8]. Besides the standard storage and retrieval operations, Galileo supports a variety of analytics functionality including range-based, exact-match, geospatial, and approximate queries. To ensure scalability, the Galileo network design is a *zero-hop* DHT similar to Apache Cassandra [2] or Amazon Dynamo [9]. This allows each *storage node* in the system to route requests directly to their destination without taking intermediary hops through the network. To further balance scalability and load distribution, nodes can be placed into *groups* (and subgroups) to create a network hierarchy.

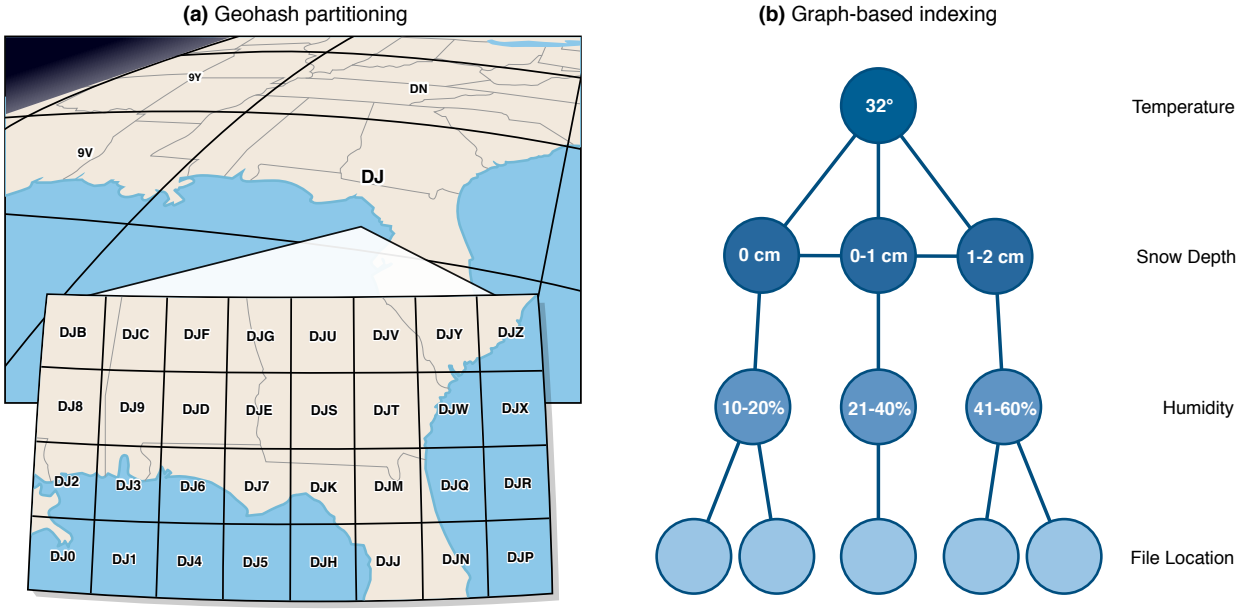

Fig. 1. An overview of key Galileo components. **(a)** demonstrates the Geohash algorithm, which represents spatial locations as one-dimensional Base-32 strings. In this example, the 1030×620 km region represented by Geohash *DJ* is divided into 32 smaller subregions by adding an additional character to the string. **(b)** Illustrates the structure of our graph-based indexes. A traversal through the graph leads to leaves that contain pointers to files on disk.

2.1 Partitioning Algorithm

We use the Geohash [10] algorithm to place incoming spatial data points into groups. Geohash describes 2D locations on Earth as one-dimensional Base-32 strings where longer strings represent smaller (and therefore more precise) spatial regions. For example, the coordinates of 30.3369 N, 81.6614 W (Jacksonville, Florida in the USA) would translate to the Geohash string *DJMUTDXVR*. Figure 1a demonstrates this concept; Jacksonville resides in the 133×155 km region described by Geohash *DJM*, which is one of 32 subregions within the region *DJ*.

In this study, we assigned our 312 virtual machines to 39 groups with 8 machines per group. Each group was assigned two spatial regions represented by two-character Geohash strings, and an SHA-1 hash of the incoming data was used to assign readings to particular nodes within the groups. This partitioning strategy has two key advantages: storage load is distributed as uniformly as possible, and the search space of queries that specify particular spatial regions can be quickly reduced to a small portion of the overall dataset.

2.2 Query Support: Metadata and Feature Graphs

While Traditional DHTs do not support queries or search functionality, Galileo incorporates a distributed, graph-based indexing structure that allows a variety of queries to be executed across its storage nodes. As multidimensional records are assimilated into the system, they are transformed into *paths* that represent a graph traversal that ends in leaf nodes with pointers to raw data blocks on disk. These paths maintain relationships between features, creating a hierarchical graph that can be reoriented, traversed, and queried. Figure 1b provides an example of a graph-based index in Galileo.

To facilitate distributed lookups, storage nodes maintain two distinct graphs: the coarse-grained and globally-distributed *feature graph*, along with a fine-grained per-node *metadata graph*. Vertices in the graph are configured to represent ranges of feature values, called *tick marks*, which control the level of quantization applied to the index. The feature graph acts as a precursor to a distributed query by reducing its search space down to the set of potential storage nodes that may contain relevant records; while a feature graph query may produce false positives, it will not produce false negatives. Once the feature graph query is complete, the distributed query is submitted to the set of relevant storage nodes for evaluation against their metadata graphs. The resulting *subgraphs* are then merged in a MapReduce-style computation and returned to the client where they can be traversed, inspected, and used to download the full-resolution data blocks from the servers.

3 Query Instrumentation and Evaluation

Queries in Galileo are expressed as MapReduce computations. The *Map* phase begins after the search space has been reduced by the feature graph and involves submitting the query to relevant storage nodes for evaluation against their respective metadata graphs. In the *Reduce* phase, resulting subgraphs are aggregated into a single result graph that will be relayed to the client. To limit network traffic during the process, storage nodes with the largest projected number of relevant results are used as Reducers. Clients are also given the option of having subgraphs streamed to them directly, a feature that is useful in applications such as visualization that can receive and process data iteratively.

3.1 Fuzzy Queries

To help locate relevant data points, Galileo supports *fuzzy queries*, which allow users to provide a set of constraints that can be relaxed to find closely related information. Our previous implementation of fuzzy queries focused on finding the closest matching data points, but simply being the closest match does not always guarantee relevance; for instance, if a user searches for flights from New York to Frankfurt departing at 3 PM and the closest match departs at 3:15 with a 48-hour layover in Iceland, then a direct flight leaving at 4 PM may be the preferable choice. Fuzzy queries are formulated in a manner similar to range or exact-match queries, but also contain flags to select which constraint(s) can be relaxed by the system.

To autonomously improve the relevance of fuzzy query results, we allow the search to expand horizontally across the graph past the nearest matching vertices. This allows the query engine to find several alternative paths that may have higher relevance. The number of files associated with each path is then totalled and the client receives a sorted list of matching data points along with information about how often they occurred in the dataset. Galileo limits fuzzy queries to $\pm 10\%$ of the known feature range, but narrower or broader limits can also be provided by the user.

3.2 Path Prediction with Dynamic Bloom Filters

Queries often result in no matching records. In such cases, it is beneficial to have fast identification of *null* queries rather than performing an unnecessary metadata graph traversal. Our previous work avoided these null queries using *path prediction* that involves each storage node maintaining a Bloom filter [11] that is updated with paths from the metadata graph as they are added to the system (excluding leaf nodes). In a federated setting, avoiding null queries becomes even more critical as latencies increase.

Since the storage nodes maintain a unique metadata graph for their specific data blocks, each Bloom filter is different as well. Bloom filters do not produce false negatives, but they do produce false positives. The false positive rate for the filter is determined by the number of: (1) bits in the filter, (2) members in the set, and (3) hash functions that are used to map an element to one of the bits in the filter.

In our previous implementation, we observed that the false positive rate increased over time as new records were added to the system. To avoid this issue, we now reinitialize the Bloom filters when their false positive rates cross a certain threshold. This involves increasing the number of the bits in the filter and number of hash functions. There are two key features in our approach: we avoid expensive disk accesses, and support continuous servicing of queries during filter replacement. Initializations do not involve disk accesses since we use the paths within the memory-resident metadata graph to populate the filter, and in our test deployment the process took about 17.3 seconds on average to complete. While the replacement filter is constructed, the old Bloom filter continues to update itself and service queries. Once the replacement filter is ready, the old filter is deleted.

4 Federated Query Evaluation

While private clouds offer a number of benefits, their primary limitation when compared to public clouds is resource capacities. In general, public clouds offer a near-infinite capacity for scaling as well as a broad range of VM configurations and pricing models. To cope with situations where increased load overwhelms the capacity of a private cloud, *cloud bursting* incorporates public cloud resources to improve scalability and overall throughput. However, our particular problem complicates cloud bursting due to its inherent storage requirements. Transferring terabytes of data in response to increased load is time-consuming, expensive (in both the monetary and computational sense), and often cannot be completed quickly enough to provide significant alleviation of load. A potential solution to this issue might involve storing compressed subsets of the data in the public cloud, but storage at scale is expensive: data transfers as well as per-gigabyte consumption costs accumulate quickly.

As an alternative approach to migrating the files themselves, we use *federated query evaluation* to push our metadata graphs to the public cloud for alleviation of the computational loads associated with resolving queries. Federated query evaluation involves four key aspects:

1) Detecting excessive loads and forecasting future query evaluation capacity based on usage trends
2) Launching VMs in the public cloud ***before*** storage nodes become overwhelmed
3) Migrating relevant indexes to the new VMs
4) Re-routing and balancing incoming query requests

These components work together to improve query throughput at the storage nodes, while also ensuring that client-side turnaround times stay low.

To evaluate our approach, we gathered real-world usage data from a busy file server at Colorado State University. We collected the data over a six-hour period before an assignment submission deadline for an undergraduate class, and then scaled and shaped the resulting dataset to model a sustained burst of query activity that would surpass the capacity of our private cloud. We chose this particular scenario for our tests because it involved frequent, iterative metadata requests along with varied read workloads. User requests were transformed to randomized queries that consisted of 50% range queries, 25% fuzzy queries, and 25% exact-match queries. Range queries included at least 5% of the overall range of values for each feature type.

Figures 2a and 2b demonstrate the query throughput observed in this test. The system is able to cope with incoming query requests until around minute 100, where the approximate computational capacity for a single VM is reached. At peak load, each storage node was responsible for around 100 clients with each issuing about 8.5 queries per second, on average. Incoming requests begin to decrease around minute 200 and fall below system capacity by minute 245. Note that for a small period of time, the number of fulfilled queries exceeds the number of requests as the storage nodes "catch up" with queries that were queued during high load. One might expect this time span to be larger, but the randomness in client requests combined with averaging across the entire system reduces the amount of visible lag exhibited on the server side.

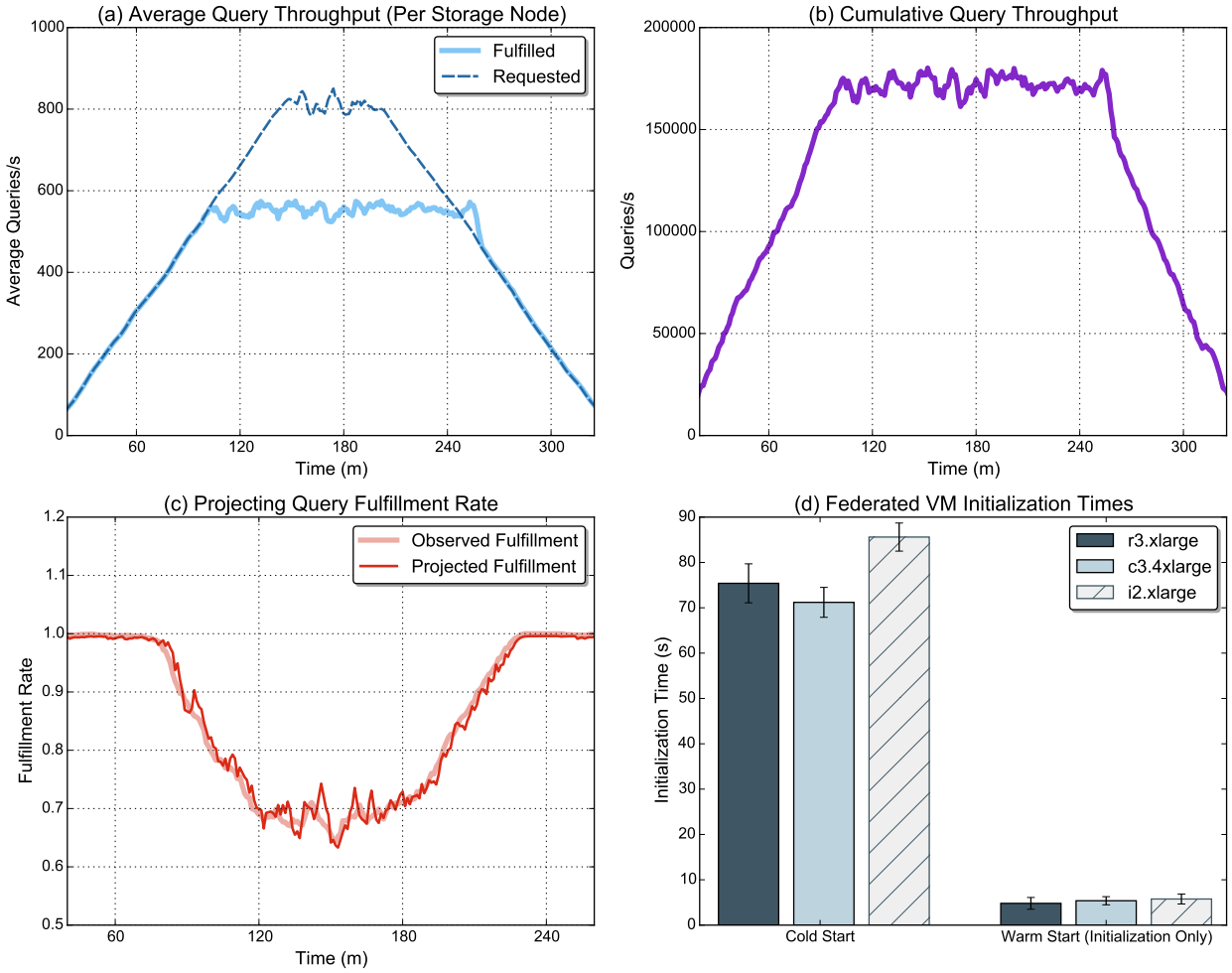

Fig. 2. Profiling different aspects of federated query evaluation: **(a)** Average per-node query throughput across the entire system during our test scenario. Each storage node could service about 550 queries per second, leading to a resource shortfall. **(b)** Cumulative query throughput across all 312 VMs. **(c)** Query fulfillment rates over time. As storage nodes become overloaded with requests, the fulfillment rate decreases. **(d)** VM startup and initialization times.

4.1 Detecting and Predicting Excessive Query Loads

We use the *fulfillment rate* to describe query capacity at a given storage node. For a particular time slice n, the fulfillment rate F is expressed as the relationship between *completed queries* (CQ) and *requested queries* (RQ):

$$F_n = \frac{CQ_n}{RQ_n}$$

When the fulfillment rate falls below 1.0 (100%), the node was unable to service all the requests and its computational query load has exceeded capacity. This results in a backlog of queries building up at the node and longer turnaround times being observed by client applications.

Determining whether or not a storage node is over capacity serves as a precursor to the cloud bursting and hybrid query evaluation process, but several other factors must be considered before the system begins launching VMs in the public cloud. One such issue is load spikes that overwhelm the capacity of a storage node for a short period of time (usually a few seconds) and then dissipate. In these cases, launching a new VM is unlikely to have any impact on query throughput. This also highlights another key issue: *reactive* cloud bursting has the potential to reduce load, but only after reduced query throughput has been detected. In applications that involve tight service-level agreements or latency-sensitive clients, cloud bursting after the fulfillment rate has decreased is already too late. For these reasons, we target *proactive* cloud bursting in our implementation.

To forecast future fulfillment rates based on usage trends, we use autoregressive integrated moving average (ARIMA) models. ARIMA models are particularly effective for time series analysis where the data in question is non-stationary; for instance, retail sales fluctuate depending on seasonality, and web services that target particular geographic regions experience different traffic patterns depending on the day of the week and time of day. ARIMA models are parameterized by three variables, p, d, and q, which correspond to the autoregressive, integrated, and moving average components of the model, respectively. We use the Hyndman-Khandakar algorithm [12] to select these parameters autonomously. Model inputs include the fulfillment rate for each time slice, the overall number of incoming and outgoing messages (storage, state transfer between nodes, administrative controls, etc.), and the number of client connections to the storage node. We also cap the fulfillment rate

at 1.0 to avoid training the model with abnormal values that occur when a node is servicing queries that were queued during high load.

Figure 2c illustrates the actual fulfillment rate observed during our benchmarks along with the projected fulfillment rate produced by our ARIMA model. For this particular evaluation, 20 minutes of training data was collected before predictions began, and forecasts were made 10 minutes into the future. If we express the fulfillment rate as a percentage, then the root-mean-square error (RMSE) was about 1.37% in this test, on average. We use the RMSE to determine prediction accuracy and to decide whether or not we are forecasting too far into the future; if the RMSE exceeds 5% then we reduce the size of the prediction window. However, determining the prediction window also depends on external factors in the public cloud.

4.2 Cloud Bursting

If the projected fulfillment rate averages below a configurable *service level* for the entirety of our prediction window, the system begins the cloud bursting process. In our test scenario, we used a service level of 95% (fulfillment rate of 0.95). The service level for a particular deployment is chosen based on the trade-off between sensitivity to changes in load and desired client-side performance. High service levels will improve query turnaround times on the client side, but may require cloud bursting to occur more frequently and are more prone to launching additional VMs during a short load spike. We also allow a hard limit to be placed on services acquired from the public cloud to ensure budget constraints are respected.

Deciding when to cloud burst depends not only on the prediction window, but also how long it takes to: (1) start the VM, (2) transfer metadata, and (3) start Galileo. While research has been conducted on how long virtual machines take to launch in Amazon's public cloud [13], the rapid hardware and software changes that occur in cloud providers make fixed values for startup times become obsolete quickly. To help predict how long a storage node will take to initialize in the public cloud, we record launch statistics in an administrative dataset within Galileo.

Figure 2d outlines the startup and initialization times (including network transfer) for three EC2 instance types: r3.xlarge (0.35 USD/hr), c3.4xlarge (0.84 USD/hr), and i2.xlarge (0.853 USD/hr). We chose these instance types because of their relatively large memory capacities (around 30 GB); if multiple storage nodes are experiencing high load, several metadata graph instances can be placed on a single VM. Besides reducing the number of VMs that must be maintained (and their accompanying hourly fees), this also helps amortize federated VM initialization times.

Launching a VM took around 80 seconds in our test scenario, resulting in a two-minute lower bound for our prediction window. Figure 3a demonstrates the effects of our cloud bursting scheme; the initial storage node handles all the requests up until minute 80, where the predicted fulfillment rate for the node falls below 0.95. A federated VM is started in the public cloud, which begins servicing query requests at minute 82 (indicated by a vertical dashed line in the figure). Incoming queries from new clients are redirected to the new VM until the original storage node can accommodate more requests. This adds a small amount of latency to the first request submitted by a new client, but allows the storage node to have fine-grained control over load balancing.

Figure 3b provides another perspective on our performance results by illustrating the client-side latency experienced when submitting queries to our private cloud and to federated VMs. In this benchmark, the *turnaround time* refers to the amount of time taken to submit a query and receive a response back from the server. As one would expect, EC2 turnaround times are higher, but the responses stay predictable and consistent throughout the test.

4.3 Maintaining Index Consistency

When duplicating indexes across distributed resources, care must be taken to ensure clients receive consistent results. Indexes residing on federated VMs are updated in an *eventually consistent* manner through a gossip protocol to avoid overwhelming the network with state updates, where nodes that issue requests most frequently will receive state updates first. Each group of storage nodes elects a leader node that receives graph state updates as a part of heartbeat messages that are sent at regular intervals for failure detection. Heartbeats pass through the group until they reach the leader node, where they are inspected to determine whether the federated VMs require a *major* or *minor* update. Minor updates represent changes to vertex metadata (such as counts or block pointers on disk), but do not involve the creation or deletion of edges or vertices. Major updates, on the other hand, are required when new vertices or edges must be added to the graph. If a major update is required, it is pushed out immediately, whereas minor updates are buffered and sent once 100 updates have accumulated or three heartbeat intervals have passed. This approach helps balance the amount of state transfers that must be completed, while also ensuring the data on the federated VMs stays fresh.

To manage the state of virtual machines in the public cloud, storage nodes acquire a lock on an administrative dataset in Galileo that describes available VMs, which nodes they are servicing queries for, and their vital statistics (load, number of cores available, memory consumption, etc.). This feature allows storage nodes to leverage available resources in the public cloud without having to launch more VMs, and enables decentralized coordination.

5 Aggregate Graph Compaction

While the main benefit behind our query evaluation strategy is higher throughput, it also provides a unique opportunity to optimize the memory footprint of our indexes. When several storage nodes are projecting high query loads, they can partition a single federated VM to share its resources. However, this approach requires clients to identify which parent storage node their query is intended for to avoid needless processing across each metadata graph instance.

A better alternative that streamlines processing and reduces memory usage is to create *aggregate metadata graphs*. Upon receiving a metadata graph from a storage node,

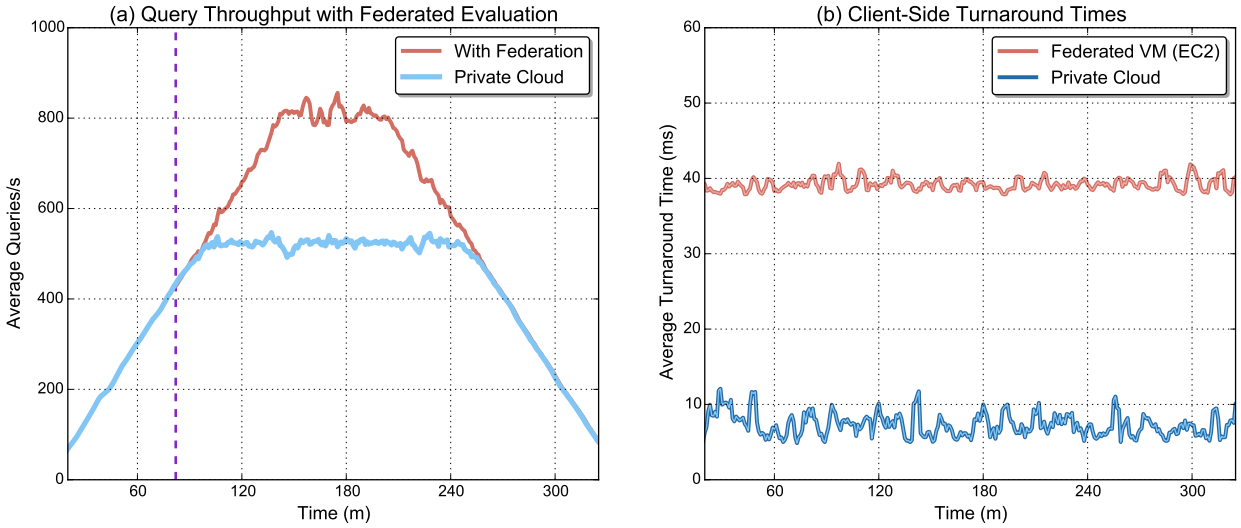

Fig. 3. **(a)** Our test scenario with federated query evaluation enabled. A federated VM is started at minute 82 (marked with a dashed vertical line) that takes on a portion of the storage load. Note that each federated VM can maintain multiple metadata graphs. **(b)** Client-side latency during the test scenario. If either the original storage node or federated VM were overwhelmed with requests, queue sizes would increase, causing client-side latency to increase significantly.

TABLE 1
Aggregate graph compaction from three different storage nodes. The nodes managing graphs A and B belong to a group serving the same spatial region (southeast USA), whereas graph C belongs to a node serving a different region (northwest USA). The reduction in vertices and edges (as a percentage of the total vertices/edges involved) is also provided for the aggregate graphs.

Graph	Vertices	Edges
A	497340	747032
B	462471	640033
C	803273	1176227
$A \cup B$	635057 (33.8%)	956846 (31.0%)
$A \cup C$	1049606 (19.3%)	1566468 (18.6%)
$A \cup B \cup C$	1122240 (36.3%)	1688159 (34.1%)

the federated VM must amend its leaves with a storage node identifier. This process makes the graph paths unique, allowing them to be mixed with paths from other metadata graphs. Due to the data volumes managed by each storage node, the likelihood of multiple graphs containing similar paths is relatively high. As a result, creating aggregate metadata graphs serves to reduce the overall number of vertices and edges required to represent the relationships between data points, while also allowing federated VMs to dedicate all of their processor cores to querying a single graph. This technique also applies to similar indexes such as the k-d tree. Table 1 contains graph statistics for three different storage nodes, two of which are part of the same group (A and B) while the third manages a completely different spatial region. When graph A is merged with either of the two other graphs, a significant reduction in both vertices and edges in the aggregate graph is achieved.

6 Conclusions

This research identifies aspects of data management frameworks that are amenable to federation in support of ad hoc analysis. It also identifies when such federation must occur and does so in a timely fashion by identifying trends and seasonality in the data. Our solution is able to perform targeted alleviation of query processing workloads and migrate to federated VMs in the order of a few seconds.

The proposed approach conserves memory within public cloud VMs by compacting metadata graphs from multiple private cluster storage nodes into a single federated VM. This allows us to reduce the number of VMs for load alleviation while also reducing monetary costs. Our gossip protocol ensures that deviations in the state of the metadata graph within the private cluster and the corresponding federated VMs are quickly resolved – we achieve this by prioritizing updates that impact accuracy of query evaluations while minimizing the network traffic needed to preserve consistency.

Acknowledgments

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018) and the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

References

[1] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser, "Cloudy: A modular cloud storage system," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1533–1536, Sep. 2010.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] T. Bicer, D. Chiu, and G. Agrawal, "A framework for data-intensive computing with cloud bursting," in *Cluster Computing, 2011 IEEE International Conference on*, Sept 2011, pp. 169–177.
[4] MongoDB Inc., "Mongodb," *http://www.mongodb.org/*, 2015.
[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.
[6] Apache Software Foundation. (2015) Apache HBase. [Online]. Available: http://hbase.apache.org
[7] M. Malensek, S. Pallickara, and S. Pallickara, "Fast, ad hoc query evaluations over multidimensional geospatial datasets," *Cloud Computing, IEEE Transactions on*, 2015.
[8] ——, "Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals," *Future Generation Computer Systems*, 2012.
[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS*, vol. 41, no. 6, pp. 205–220, 2007.
[10] G. Niemeyer. (2008) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash
[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[12] R. J. Hyndman and Y. Khandakar, "Automatic time series forecasting: The forecast package for R," *Journal of Statistical Software*, vol. 27, no. 3, pp. 1–22, 7 2008.
[13] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Cloud Computing, 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 423–430.

Matthew Malensek is a Ph.D. student in the Department of Computer Science at Colorado State University. His research involves the design and implementation of large-scale distributed systems, data-intensive computing, and cloud computing. Matthew received his Masters degree in Computer Science from Colorado State University.

Sangmi Pallickara is an Assistant Professor in the Department of Computer Science at Colorado State University. Her research interests are in the area of large-scale scientific data management, data mining, scientific metadata, and data-intensive computing. She received her Masters and Ph.D. degrees in Computer Science from Syracuse University and Florida State University, respectively.

Shrideep Pallickara is an Associate Professor in the Department of Computer Science at Colorado State University. His research interests are in the area of large-scale distributed systems, specifically cloud computing and streaming. He received his Masters and Ph.D. degrees from Syracuse University. He is a recipient of an NSF CAREER award.