Synopsis: A Distributed Sketch over Voluminous Spatiotemporal Observational Streams

Thilina Buddhika, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara, *Member, IEEE*

Abstract—Networked observational devices have proliferated in recent years, contributing to voluminous data streams from a variety of sources and problem domains. These streams often have a spatiotemporal component and include multidimensional *features* of interest. Processing such data in an offline fashion using batch systems or data warehouses is costly from both a storage and computational standpoint, and in many situations the insights derived from the data streams are useful only if they are timely. In this study, we propose Synopsis, an online, distributed *sketch* that is constructed from voluminous spatiotemporal data streams. The sketch summarizes feature values and inter-feature relationships in memory to facilitate real-time query evaluations and to serve as input to computations expressed using analytical engines. As the data streams evolve, Synopsis performs targeted dynamic scaling to ensure high accuracy and effective resource utilization. We evaluate our system in the context of two real-world spatiotemporal datasets and demonstrate its efficacy in both scalability and query evaluations.

Index Terms—Data sketches, streaming systems, spatiotemporal data, query evaluations

1 Introduction

The proliferation of remote sensing equipment such as radars and satellites, networked sensors, commercial mapping, location-based services, and sales tracking applications have resulted in exponential growth of spatiotemporal data. Such datasets comprise observations where both the location and time of measurement are available in addition to *features* of interest (such as humidity, air quality, disease prevalence, sales, etc.). This information can be leveraged in several domains, including atmospheric science, epidemiology, environmental science, geosciences, smart cities, and commercial applications. In these settings, queries over the data must be *expressive* and execute in real time, regardless of data volumes.

Spatiotemporal datasets are naturally multidimensional with multiple features of interest being reported/recorded continuously for a particular timestamp and geolocation. The values associated with these features are continually changing; in other words, the dataset *feature space* is always evolving. Queries specified over these datasets may have a wide range of characteristics encompassing the frequency at which they are evaluated and their spatiotemporal scope. The crux of this paper is to support query evaluations and data processing over continually-arriving observational data. We achieve this via construction of an in-memory distributed *sketch* that maintains a compact representation of the data. The sketch is also an effective surrogate for the data that it snapshots and serves as input for computations.

Challenges. Support for real-time evaluation of queries and analysis over a feature space that is continually evolving introduces unique challenges. These include:

- *Data volumes and arrival rates:* It is infeasible to store all observations, which may arrive continually and at high rates. This is especially true if the arrival rates outpace disk speeds.
- *I/O Costs:* Memory accesses are 5-6 orders of magnitude faster than disk accesses. Given the data volumes, disk accesses during query evaluations or analysis are infeasible.
- *Accuracy:* Queries evaluations must be accurate, with appropriate error bounds included in the results.
- *Spatiotemporal characteristics:* Queries and analysis may target both spatial and chronological properties of the dataset.

Research Questions. The challenges associated with accomplishing this functionality led us to formulate the following:

RQ-1: How can we generate compact, memory-resident representations of the observational space while accounting for spatiotemporal attributes? The resulting *sketch* must be amenable to fast, continuous updates to ensure its representativeness and fidelity to the original data.

RQ-2: How can we scale effectively in situations where system load is high or observations arrive faster than the sketch can be updated? The density and arrival rates for observations may vary based on geospatial characteristics; for example, New York would have a far higher rate of observations than Denver.

RQ-3: How can we enable expressive, low-latency queries over the distributed sketch while also maintaining accuracy? Given that the sketch is a compact representation of the data, queries facilitate high-level analysis without requiring users to understand the underlying system implementation.

- *The authors are with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: {thilinab, malensek, sangmi, shrideep}@cs.colostate.edu.*

Manuscript received 23 Feb. 2017; revised 10 July 2017; accepted 25 July 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Thilina Buddhika.)
Recommended for acceptance by J. Xu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2734661

Approach Summary. Similar to other sketches, the design of SYNOPSIS was guided by its desired functionality. SYNOPSIS is a compact, effective surrogate for voluminous data; the system extracts metadata from observations and organizes this information to support relational queries targeting different portions of the feature space. We support selection, joins, aggregations, and sorting. The SYNOPSIS sketch can interoperate and provide input data to general purpose computations expressed using popular analytic engines such as Spark [1], [2], TensorFlow [3], [4], Hadoop [5], [6], [7], and VW [8].

Our sketch is also naturally amenable to distribution, with each machine in the cluster holding information about a particular subset of the observational space. This ensures that each cluster-node can evaluate multiple concurrent queries independently. The sketch is capable of scaling in or out depending on streaming ingress rates and memory footprints, with scale-out operations that support targeted alleviation of hotspots. SYNOPSIS manages the complexity of identifying these hotspots, splitting portions of the sketch, and migrating relevant subsets. Distributing the sketch allows us to maintain a finer-grained representation of the feature space while also improving the accuracy of query evaluations; e.g., an arctic region and a tropical region would be maintained on separate nodes that specialize for particular climates.

Paper Contributions. To our knowledge, SYNOPSIS is the first sketch tailored specifically for spatiotemporal observational data. The methodology of this study is centered around our novel in-memory data structure, *SIFT* (Section 3.2.1), which employs a hierarchical forest-of-trees approach combined with online, running summary statistics to compactly represent observational data. In addition to the memory benefits of the SIFT, the data structure is amenable to distribution across a cluster of machines. This allows SYNOPSIS to cope with high-rate data arrivals and scale dynamically with changes in problem size or resource availability. Both dynamic scaling and querying are facilitated by efficient tree-based lookup operations. For analytic tasks, the SIFT acts as an effective surrogate for full-resolution observations, enabling expressive queries over arbitrary spatiotemporal scopes, generation of synthetic datasets, and interoperation with popular analytical engines.

Paper Organization. Section 2 provides a system overview, followed by methodology in Section 3. A performance evaluation is presented in Section 4, while Section 5 demonstrates applications of SYNOPSIS, Section 6 discusses related approaches, and Section 7 concludes the paper.

2 SYSTEM OVERVIEW AND PRELIMINARIES

SYNOPSIS is a distributed sketch constructed over voluminous spatiotemporal data streams. The number of sketchlets (executing on different machines) that comprise the distributed sketch varies dynamically as the system scales in or out to cope with data arrival rates and memory pressure. SYNOPSIS assimilates, organizes, and compacts spatiotemporal data streams that comprise the sketch. A stream partitioning scheme, based on the Geohash algorithm, is used to route packets to the appropriate sketchlet. Sketchlets process stream packets emitted by stream ingesters and construct compact, in-memory representations of the observational data by extracting metadata from stream packets. During dynamic scaling, the geographic extents managed by a sketchlet vary.

System Components. SYNOPSIS leverages the Neptune stream processing system [9], [10] and relies on a set of auxiliary services that are needed to construct, update, and maintain the sketch, as well as adapt to changing system conditions:

Control plane is responsible for orchestrating control messages exchanged between sketchlets as part of various distributed protocols such as dynamic scaling. It is decoupled from the generic data plane to ensure higher priority and low latency processing without being affected by buffering delays and backpressure experienced during stream processing.

Gossip subsystem is used by the sketchlets to gossip about their state periodically (based on time intervals and the number of pending updates) as well as when a change in state occurs to establish an approximate global view of the system. SYNOPSIS supports *eventual consistency* with respect to these updates given their propagation and convergence delays.

Querying subsystem is responsible for the distributed evaluation of queries. This involves forwarding queries to relevant sketchlets; in some cases, multiple sketchlets may be involved based on the geographical scope of the query.

Monitoring subsystem probes sketchlets comprising SYNOPSIS periodically to gather metrics that impact performance of the system. These include memory utilization and backlog information based on packet arrival rates and updates to the in-memory structures. This information is used for dynamic scaling recommendations as explained in Sections 3.3 and 3.4.

Data Model. While designing SYNOPSIS, we targeted a data model wherein observations are geotagged and have chronological timestamps indicating where and when the observations were made. Location information is encoded as $\langle latitude, longitude \rangle$ tuples. Observations contain multiple features (temperature, humidity, wind speed, etc.), and may be encoded as $\langle feature\_name, value \rangle$ tuples or may have predefined positions within the serialized data representation.

Query Support. SYNOPSIS supports common query constructs such as selects or joins, while also providing rich analytical queries that report statistical information, make predictions, or produce synthetic datasets (detailed fully in Section 3.5). A key innovation in our query support is that portions of the sketch itself can be retrieved and manipulated by clients. The following query demonstrates this functionality, where climate features are requested from a region when the wind speed is more than a standard deviation away from the mean:

```
SELECT location, precipitation, humidity
WHERE location LIKE 'dj%' AND
  (wind_sp > MEAN(wind_sp) + STD(wind_sp)
  OR wind_sp < MEAN(wind_sp) - STD(wind_sp))
```

Stream Partitioning. We use the Geohash algorithm [11] to balance load and partition incoming data streams. Geohash divides the earth into a hierarchy of bounding boxes identified by Base 32 strings; the longer the geohash string, the more precise the bounding box. Fig. 1 illustrates this hierarchy. Most of the eastern United States is contained within the bounding box described by geohash *D*, while *DJ* encompasses substantial parts of Florida, Georgia, and Alabama. The bounding box *DJKJ* (highlighted in red) contains Tallahassee, Florida. This hierarchical representation enables SYNOPSIS to cope with both low- and high-density regions: several sketchlets may be tasked with managing streams

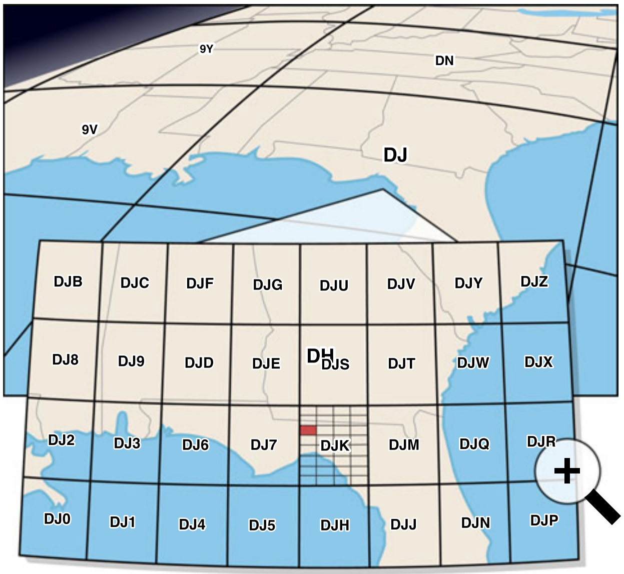


Fig. 1. Visual demonstration of the Geohash algorithm.

originating in and around large cities, while rural areas fall under the purview of a single node.

Each bit added to a geohash string reduces its scope by half, with each character represented by five bits ($2^5 = 32$). In other words, a four-character geohash string represents 20 spatial subdivisions applied recursively to each resulting region. This property allows us to manage and allocate resources across a variety of observational densities.

3 Methodology

In this section, we discuss the construction of the distributed sketch, sketchlet data structure, dynamic scaling, and query support in Synopsis. We report microbenchmark results which were run using a single machine (HP DL160; Xeon E5620; 12 GB RAM) demonstrating the efficacy of individual aspects of the system. Our input data was sourced from NOAA North American Mesoscale (NAM) Forecast System [12].

3.1 Sketch

From a macroscopic view, the sketch is organized as a distributed prefix tree. All descendant nodes—*sketchlets*—are responsible for a particular geospatial scope and share a common prefix with their parent. One of our primary goals behind Synopsis is to ensure the sketch is performant, flexible, and amenable to scaling. The sketch initiates scale-out operations to relieve memory pressure and preserve performance, while scale-in operations conserve memory. Further, any sketchlet may serve as the *conduit* for queries or analytic operations.

The geohash algorithm plays a central role in the organization of the distributed sketch. Since locations are represented by bounding boxes, the algorithm facilitates collocation of observations from particular geographical scopes. This allows the conduit to redirect queries effectively and ensure data locality. Increases in the length of the geohash string correspond to geographically smaller (and more precise) bounding boxes. This is well-aligned with dynamic scaling performed by the sketch to manage memory requirements. Scaling operations within the sketch are targeted; scaling out targets geospatial locations with increased density of observations to relieve memory pressure and alleviate performance bottlenecks, while scaling in targets geolocations where there is a sparsity in available data to conserve memory.

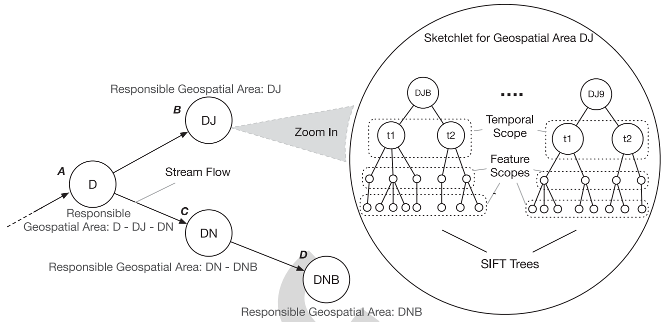


Fig. 2. A demonstration of the distributed sketch for geohash prefix D. The sketchlets for prefixes DJ and DN have scaled out due to high volume of observations. Each sketchlet maintains a SIFT, with each tree responsible for a geospatial subregion.

Each sketchlet is responsible for real-time organization, summarization, and compaction of observational data from the geographical scope represented by its geohash. The sketchlet performs two operations: first, it extracts metadata from incoming observations, including geolocations, chronological information, and features encapsulated within the observation. Second, the sketchlet is responsible for summarization and compaction of the extracted features.

The sketchlet organizes its summarization in a data structure called SIFT (Summarization Involving a Forest of Trees). The edges and vertices within each SIFT tree maintain inter-feature relationships, while leaves contain online, in-memory summary statistics and correlation information to support statistical queries and generation of synthetic datasets. The number of edges at each level within the subtrees corresponds to density-based dynamic binning of a particular feature to reduce error. The underlying principle within this data structure is *grouping* to exploit similarities in values reported within observations; grouping allows us to preserve fidelity of the observational space while conserving memory.

A simplified version of the distributed sketch for geospatial region *D* is depicted in Fig. 2. Each tree within the SIFT is rooted at a higher precision geohash than that associated with the sketchlet. For example, at a sketchlet with a geohash prefix, *DJ*, the trees within the SIFT at that sketchlet are rooted at higher precision geohashes such as *DJB*, *DJC*, *DJF*, etc. An advantage of this approach is that the sketchlet partitions data from the overall geospatial scope into smaller regions, further improving the grouping of observations.

The second level of the SIFT is used to group observations based on their temporal properties. This approach allows us to exploit similarity in readings reported for a particular time range. Note that as the trees are traversed, this organization strategy means that all descendants of a temporal node correspond to measurements reported for a particular region and for a particular temporal scope. The SIFT data structure also supports finer-grained temporal resolutions for the recent past—e.g., minutes, hours, day, weeks, etc.—along with targeted compaction operations that fold finer-grained temporal scopes into a coarser grained scopes as time advances. Specifically, our organizational structure allows us to support varying levels of expressiveness for different temporal scopes, with recent observations being represented more expressively.

The grouping concept also extends to individual features. Each feature occupies a level within an individual tree in

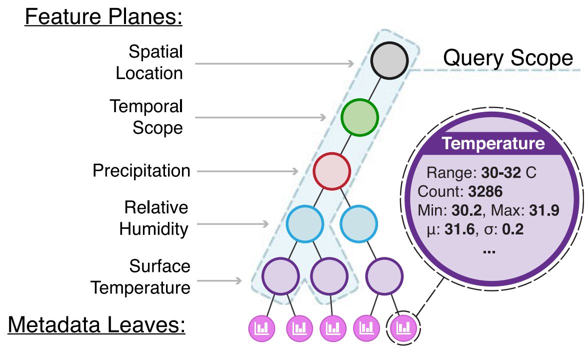


Fig. 3. A simplified SIFT tree with five planes and a sample query scope. In production settings, these trees contain hundreds of thousands of vertices and edges.

SIFT. At each level, the range of values that a feature can take is broken up into a set of bins (corresponding to the range of values) that they take. These ranges are determined using kernel density estimation (KDE) to ensure that the binning of features is representative of the observed density in the distribution of values for that feature at the particular spatiotemporal scope. Each node (or bin) maintains the min, max, standard deviation, mean, and the total number of observations it is responsible for. This is useful during the creation of synthetic datasets that are representative of the observational space for a particular spatiotemporal scope.

The methodology behind SIFT accomplishes two key objectives: first, it captures the distribution of feature values across a spatiotemporal scope. Second, it supports targeted reductions in the observational data volumes while being representative of the observed feature values. This is in contrast to a random sampling scheme, which may be unable to recreate distributions with high fidelity for arbitrary spatiotemporal scopes.

The organization of the sketchlet is such that it is amenable to scale-out and scale-in operations of the distributed sketch. A key feature provided by the SIFT data structure is support for scaling operations. For example, if a subregion represented by a tree within the forest maintained at each sketchlet has a higher density (and variability) of the reported observational values, that tree would have a correspondingly higher memory footprint within the data structure. This allows us to target scaling maneuvers to particular subregions managed at a sketchlet to alleviate memory pressure. During scale-in operations, descendants can be folded into the parent; the descendant's SIFT is simply added as a tree to the SIFT maintained at the parent.

Systems View of the Sketch. The SYNOPSIS sketch, comprising sketchlets dispersed over multiple machines, is a compact and memory-resident surrogate for the entire observational space. The sketch may be used for any purpose that regular, on-disk data is used for including but not limited to query evaluations, assessing statistical properties of the data, and launching computations using well-known analytical engines. The sketch is adaptive and evolves over time, with the number of sketchlets varying as scaling maneuvers occur to cope with data volumes and memory management. The structure of the SIFT also varies over time as temporal scopes are aggregated, features binned, and scaling occurs.

3.2 Sketchlet

Sketchlets maintain compact, multidimensional, tree-based representations of incoming data streams in the SIFT data structure. Each in-memory SIFT can be queried to retrieve statistical properties about the underlying data or discover how features interact. Due to the voluminous nature of these data streams, storing each record in main memory is not practical. Therefore, the queries supported by our framework are facilitated by compact, online metadata collection and quantization methods. These techniques ensure high accuracy while also conforming to the memory requirements of the system. To further improve accuracy, we bias our algorithms toward the most recent data points while reducing the resolution of the oldest.

3.2.1 *SIFT Structure*

SIFT instances are maintained as hierarchical trees with feature values stored in the vertices. Each level of the hierarchy, called a *plane*, represents a particular data type, and traversing through vertices in this feature hierarchy incrementally reduces the search space of a query. Upon insertion of a multidimensional observation, each feature is arranged to form a *path* through the tree and added based on the current feature hierarchy. Paths taken through the tree during a lookup are influenced by the specificity of the query, with additional feature expressions constraining the *query scope*; an empty query would result in a scope that spans the entire tree. Fig. 3 demonstrates the structure of a SIFT tree and highlights a query and its scope. SIFT trees are internally arranged as modified red-black trees, resulting in lookup and insertion complexities of $O(log\ n)$.

Metadata records for paths through the feature hierarchy are stored at leaf nodes. Each record contains statistics that are updated in an online fashion using Welford's method [13]. Welford's method maintains the number of observations, n, the running mean, $\bar{x}$, and the sum of squares of differences from the current mean, S_n, as in the following recurrence relation,

$$\begin{aligned}
&\bar{x}_0 = 0, S_0 = 0 \\
&\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \\
&S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n).
\end{aligned}$$

Besides the observation count and running mean, this enables calculation of the variance and standard deviation of the observed values: $\sigma^2 = \frac{S_n}{n}; \sigma = \sqrt{S_n/n}$. Our implementation of Welford's method also maintains the sum of cross products between features to track cross-feature relationships, such as the correlation between temperature values and humidity. Leaf nodes may also be *merged* to combine their respective summary statistics into a single aggregate summary, which allows queries to be evaluated across multiple sketchlets and then fused into a single, coherent result.

3.2.2 *Structural Compaction*

The number of unique feature types stored in the SIFT directly influences the size of the hierarchy, which impacts memory consumption. However, memory use can be managed by manipulating the hierarchical configuration of the trees to increase vertex reuse. In general, features that exhibit high *fan-out* (many outgoing edges leading to the next level of the hierarchy) should be placed near the bottom of the tree to reduce its overall size. Listing 1 demonstrates how the fan-out score is calculated.

Listing 1. Calculation of the fan-out score (average number of outgoing edges) for dynamic reconfiguration.

```
def fan_out_score(feature):
  sc = Synopsis.context()
  # Select all vertices for this feature
  vertices = sc.query('SELECT ' + str(feature))
  edges_out = 0
  for vertex in vertices:
    edges_out += vertex.num_neighbors()
  fan_out = edges_out / len(vertices)
  return fan_out
```

To illustrate this concept, consider both a boolean feature and spatial location being used as the root of the tree. With the boolean feature, two possible partitions of the tree are created. However, using the spatial location leads to the creation of hundreds or thousands of subtrees, depending on the geohash resolution being used at the sketchlet. This leads to low vertex reuse. Consequently, we *compact* the logical representation of the SIFT by aggregating vertices from the entire forest and reorienting the planes to conserve memory. Listing 2 presents the feature plane compaction algorithm.

Listing 2. Feature plane compaction algorithm; the hierarchy is reconfigured based on sorted fan-out scores.

```
def compact_hierarchy():
  sc = Synopsis.context()
  scores = []
  for feature in sc.features:
    fan_out = fan_out_score(feature)
    scores.append((fan_out, feature))
  new_hierarchy = sort_ascending(scores)
  return new_hierarchy
```

One notable result of this process is that vertices near the bottom of the hierarchy may be responsible for storing spatial locations of the data points rather than the root of the tree as depicted in our conceptual model of the SIFT. After reconfiguration, the fan-out scores are leveraged to estimate the lower and upper bounds for memory usage (the reversed configuration will yield the highest memory consumption).

3.2.3 Density-Driven Quantization

Maintaining data points, statistics, and cross-feature relationships in memory at full resolution is infeasible when faced with voluminous datasets, even when load is balanced over several computing resources. To reduce the memory consumption of SIFT instances we perform *quantization*—targeted reduction of resolution—which allows vertices in the tree to be merged, thus enabling single vertices to represent a collection of values. We determine which vertices should be merged by splitting each range of feature values into a configurable number of *bins*. After quantization, each vertex represents a range of observations.

To determine the size and quantity of these bins, trees within the SIFT maintain additional metadata provided by the multivariate online kernel density estimation (oKDE) algorithm developed by Kristan et al. [14]. While it is possible to recompute kernel density estimates periodically for each feature type using in-memory samples [15], the online approach afforded by oKDE requires less overall CPU usage and memory, which is crucial in streaming environments.

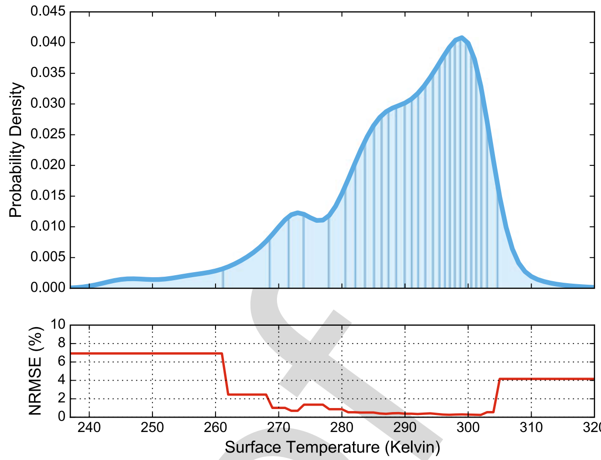


Fig. 4. Quantized surface temperatures, with 29 vertex bins. Each bin is indicated by a vertical line under the curve.

oKDE assimilates data incrementally at runtime to create a dynamic probability density function (PDF) for each feature type. The smoothing parameter used to create the PDF, called the *bandwidth*, is selected autonomously using Silverman's rule [16]. Silverman's rule assumes that data tends to follow a normal distribution, which is generally true for naturally-occurring observations. However, we also allow the smoothing parameter be selectively reconfigured for different problem types.

During the quantization process, these PDFs are used to ensure that each bin is assigned an approximately equal proportion of the feature density to create small, highly-accurate bins, while the overall number of bins is influenced by memory availability. Given a PDF and bin count for a feature, our quantization algorithm iterates through the bins and assigns them each an equal portion of the probability density. This process has a time complexity of $O(n)$ where n is the number of bins, but it is worth noting that n is generally small (around 30 on average) and the algorithm is not run frequently.

Fig. 4 illustrates the quantization process for the *surface temperature* feature in our atmospheric test dataset [12]: the highest densities of values are stored in the smallest bins (indicated by vertical lines under the curve), improving overall accuracy. To evaluate accuracy, we compare the mean values of each bin with the actual, full-resolution data points. Consequently, the *standard error* ($\sigma_{\bar{x}}$) can be calculated from our running summary statistics to judge the accuracy level of the bins based on how well they represent the mean: $\sigma_{\bar{x}} = \sqrt{S_n/n^2}$. This information is provided alongside any query results returned by the system. During initialization, we calculate the normalized error for each data point empirically (shown in the lower portion of Fig. 4). For values that are observed less frequently, the error rate is higher; temperatures from 240-260 Kelvin (-33.15 to $-13.15°$ C) reach a normalized root-mean-square error (NRMSE) of about 7 percent. However, approximately 80 percent of the values in the tree will be assigned to vertices with an error of about 0.5 percent. In practice, this means that commonly-observed values will be within 0.25 Kelvin of their actual value.

Table 1 compares full-resolution and quantized trees generated from a month of data with 20 unique features, which include atmospheric information such as temperature, humidity, precipitation, and cloud cover. In this configuration, our quantization algorithm reduced memory

TABLE 1
Tree Statistics before and after Our Dynamic Quantization Algorithm over One Month of Ingested Data

Metric	Original	Quantized	Change
Vertices	3,104,874	1,238,424	-60.1%
Edges	3,367,665	1,441,639	-57.2%
Leaves	262,792	203,216	-22.7%
Memory	1,710.6 MB	643.1 MB	-62.4%

consumption by about 62.4 percent, which allows much more historical data and larger geographical areas to be maintained in each SIFT.

3.2.4 Temporal Dimensionality Reduction

While our quantization approach enables Synopsis to retain large volumes of data in main memory, we also offer a temporal *accuracy gradient* to ensure the most relevant data points are prioritized for high accuracy. This is achieved by iteratively removing tree paths from the SIFT hierarchy in the oldest subtrees, eventually phasing out old records. As data ages, this process results in the creation of temporal accuracy bands.

Selective temporal dimensionality reduction proceeds in a bottom-up fashion, starting from the bottom of the hierarchy. Given a set of relevant vertices, neighboring bins are merged uniformly across the feature space. As the bins are merged, their respective metadata is also merged, reducing memory consumption. Given two metadata instances, merging results in half the memory footprint. However, it is worth noting that this process is irreversible; once metadata has been merged, it cannot be split at a later point in time. As time passes, entire portions of the feature space are compacted until a single metadata record is left for a particular temporal range. This allows users to still query the summary statistics and models for historical data, but at a lower level of accuracy.

3.2.5 Facilitating Scalability

The SIFT is designed to facilitate distribution across several computing resources. Specifically, splitting and merging functionality is exposed via the same query interface used for resolving user requests. Queries proceed in a depth-first fashion, locating a *subtree root* within the hierarchy. Once the root is established, either: (1) its descendants are serialized for transmission to another sketchlet, or (2) an incoming subtree is merged in place. During merges, existing paths are reused with only leaf nodes requiring updates.

3.3 Coping with High Loads: Scaling Out

There are two primary approaches to scaling a sketchlet that is experiencing high load: *replication* and *load migration*. In replication-based scaling, new sketchlets are spawned during high data arrival rates that are responsible for identical spatial scopes as their originating sketchlet. Assimilation of the newly-created sketchlet involves partitioning inbound streams directed to the original sketchlet. The upstream node (e.g., stream ingester) is responsible for partitioning, which may be performed in a skewed fashion with the new sketchlet receiving a larger portion of the inbound stream. Alternatively, inbound streams to the original sketchlet may also be partitioned in a round-robin fashion between the original and the newly-created sketchlet. Using a replication-based scaling with a round-robin style stream partitioning is memory inefficient because of the possibility of multiple SIFT trees with significantly overlapping sets of vertices and edges. Alternatively, *targeted* load migration selects geospatial scopes that are experiencing heavy load; both data arrival and SIFT update rates are considered when deciding which trees to migrate.

In Synopsis, we use targeted load migration for scaling out. Our implementation closely follows the MAPE loop [17] which comprises four phases: monitor (M), analyze (A), planning (P) and execution (E). A *monitoring* task within every sketchlet periodically gathers two performance metrics:

1) *Length of the backlog:* This represents the number of unprocessed messages in the queue. If the sketchlet cannot keep up with the incoming data rate, the backlog grows.
2) *Memory pressure:* Each sketchlet is allocated a fixed amount of memory. Exceeding these memory limits creates memory pressure causing extended garbage collection cycles and increased paging activity, eventually leading to reduced performance. The monitoring task continuously records memory utilization and triggers scaling activities.

The objective of scaling out is to maintain *stability* at each sketchlet. We define stability as the ability to keep up with incoming data rates while incurring a manageable memory pressure. During the *analyze* phase, we use threshold-based rules [18] to issue scale-out recommendations to sketchlets, which are issued if *either* of the following rules are consistently satisfied for a certain number of monitoring cycles:

- Backlog growth, which indicates that a portion of the load needs to be migrated to a different sketchlet.
- High overall memory utilization above a threshold, which is usually set below the memory limits to allow a capacity buffer for the process to avoid oscillation.

Upon receiving a scale out recommendation during monitoring, the sketchlet executes the *planning* and *execution* phases.

During the planning phase, the sketchlet chooses portion (s) of the region within its current purview, i.e., a set of SIFT trees, to be handed over to another sketchlet. For this task, it relies on performance metrics it maintains for each subregion and a numeric value provided by the scale-out recommendation that measures how much load should be migrated. These metrics includes the data rate and the memory consumption for each subregion. If the backlog growth based threshold rule has triggered the scale out operation, the subregion metrics are sorted based on their update rates in the descending order. Otherwise they are sorted based on their memory consumption. Then a simple bin-packing algorithm is used to choose a minimum set of subregions for migration such that the excess load is removed from the current sketchlet.

Only a single scaling operation takes place at a given time per sketchlet, which is enforced by a mutex lock. Further, every scaling operation is followed by a *stabilization period* where no scaling operation takes place and system does not enter the monitoring phase for the next MAPE cycle. The objective of these constraints is to avoid oscillations in scaling activities; for instance, repetitively scaling out in the presence of memory pressure could result in overprovisioning, which would then lead to recurring scale-in operations.

Fig. 5 depicts the phases of the scale-out protocol with respect to our example in Fig. 2 when sketchlet C is scaling out to sketchlet D. Once the sketchlet decides on subregions

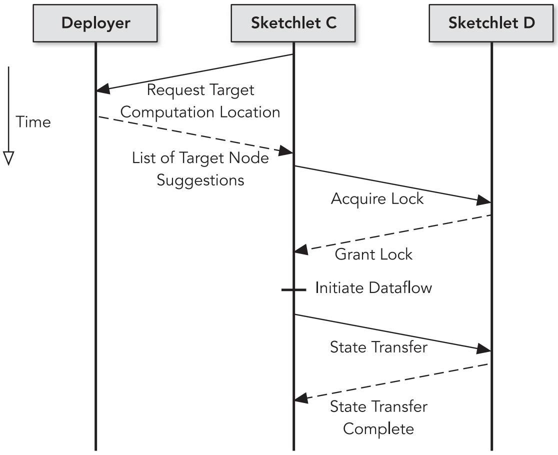


Fig. 5. Scale-out protocol.

to scale, it initiates the scale-out protocol by contacting the *deployer* process, which is responsible for launching tasks. In this message, it includes a list of preferred target sketchlets for the load migration as well as memory requirements and expected message rate for the load. The preferred sketchlet set includes the sketchlets that already hold other subregions. It minimizes the number of sketchlets responsible for each geographical region to reduce communication during query evaluations. The SYNOPSIS deployer component has an approximate view of the entire system constructed through gossip messages. This includes the memory pressure and cumulative backlog information for each sketchlet. Based on this view and the information present in the request, the deployer replies back with a set of candidate target sketchlets. Only if a suitable candidate cannot be found from the set of current sketchlets will a new sketchlet be spawned. Upon receiving a response from the deployer, the sketchlet (parent) contacts the target sketchlet (child) and tries to acquire the mutex. A lock will be granted only if the target can accommodate the load and no other scaling operations are taking place. If the lock acquisition fails, another candidate from the list is attempted; otherwise, the parent sketchlet will create a pass-through channel and direct traffic corresponding to migrated regions towards the child sketchlet. Once this process is complete, the parent sketchlet will initiate a state transfer asynchronously using a background channel to ensure the stream data flow is not affected, and update the child sketchlet's memory utilization metrics to account for the pending state transfer.

As the data flow tree grows with scale-out operations, having parent sketchlets pass traffic through to their children becomes inefficient because of higher bandwidth consumption as well as increased latency due to the additional network hops the packets have to traverse through. To circumvent this, we allow *short circuiting*, which redirects traffic from stream ingesters straight the downstream sketchlets. For instance, stream ingesters will send data directly to sketchlet D using the short circuited route bypassing sketchlets A and C in Fig. 2. We use our gossiping subsystem to update parent sketchlets about the child's performance metrics required for scaling in (Section 3.4).

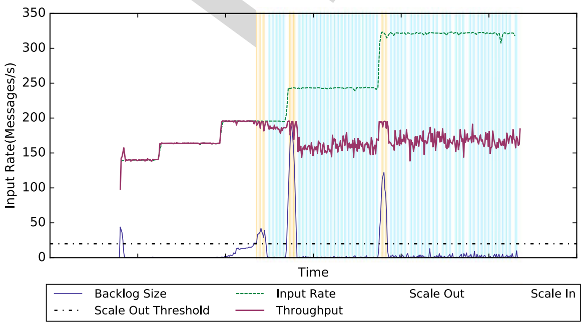
We evaluated how each of these rules triggers dynamic scaling activities to maintain the system stability. For this experiment, we have enabled only a single threshold-based rule at a time to demonstrate its effectiveness. To evaluate the backlog based threshold rule, we captured how backlog length and throughput at an individual sketchlet varies with the input rate. The sketchlet immediately received data from stream ingesters, hence the input rate observed at the sketchlet closely resembled the varying data ingestion rate. As shown in Fig. 6a, scaling out helps a sketchlet to keep pace with the variations in the workload, which in turn causes the backlog to stay within a safe range. This benchmark also shows infrequent, rapid scale-out and continuous, gradual scale-in as explained in Section 3.3.

Fig. 6b demonstrates how memory consumption threshold rules trigger scaling maneuvers to maintain the stability of an individual sketchlet. We have used 0.45 of the total memory available to a JVM process as the upper threshold for triggering scale-out operations. In certain occasions, it is required to perform multiple consecutive scaling out operations (interleaved with the cooling down periods) to bring memory usage to the desired level due to the increased utilization caused by background data ingestion.

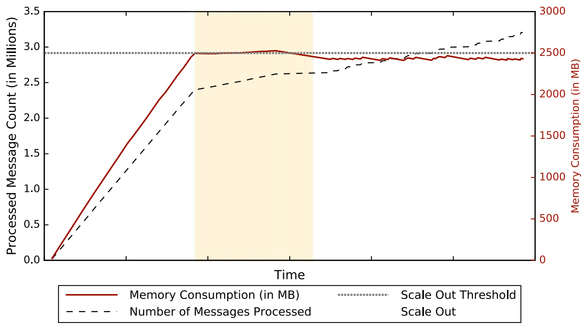
3.4 Scaling In: Conserving Resources

During scaling in, sketchlets merge scaled-out subregions back into their SIFT. This ensures better resource utilization and reduces the number of sketchlets contacted during query evaluations. Scaling in is also guarded by the same mutex used for scaling out and is also followed by a stabilization period.

Monitoring and analysis during scale-in operations proceeds similarly to scaling out, except for the obvious change to the threshold-based rules: now *both* memory pressure and backlog length metrics should consistently record values below a predefined lower threshold. When scaling in, we use a less aggressive scheme than scaling out; a single subregion is acquired during a single scale-in operation. Scaling in is more complex because it involves more than one sketchlet in



(a) Triggered by backlog growth based threshold rules



(b) Triggered by memory usage based threshold rules

Fig. 6. Scaling out based on backlog growth and memory usage enables maintaining stability at an individual sketchlet.

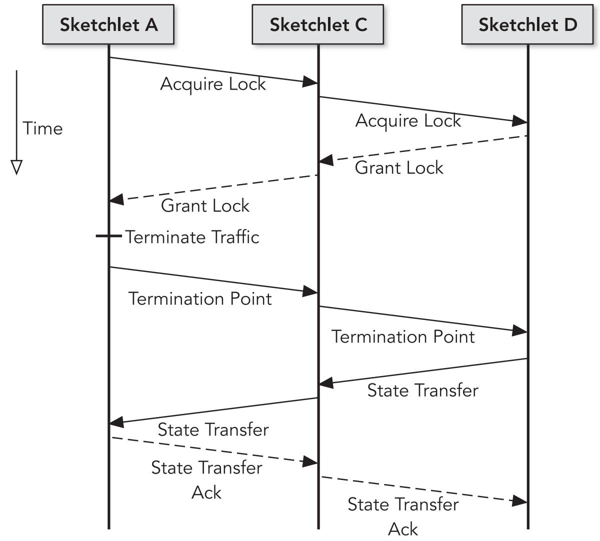


Fig. 7. Scale-in protocol.

most cases; at this point, it is possible that further descendant scale-out operations have taken place. For instance, if sketchlet A in Fig. 2 decides to scale in subregion *DN*, then it must communicate with sketchlets C and D.

The scale-in protocol starts with a lock acquisition protocol similar to the scaling out protocol, but involves locking the entire subtree. The steps are depicted in Fig. 7 with respect to our example in Fig. 2 where sketchlet C is scaled in. As per our example, sketchlet A will acquire locks for both sketchlets C and D. Locks are acquired in a top-to-bottom fashion where parent locks itself and then attempts to lock the child. If lock acquisition fails for any part of the subtree, the scale-in operation is aborted and the monitoring process starts the next iteration of the MAPE loop immediately. If the lock acquisition is successful, then data flow to the child sketchlet corresponding to this subregion is immediately terminated.

The state acquisition phase begins next. To ensure that Synopsis does not lose any messages, the initiating sketchlet sends a *termination point* control message to the child sketchlet. The termination point is the sequence number of the last message sent to the child sketchlet either by the parent itself or by the short circuit channel. Once the child sketchlet has processed every message up to the termination point, it sends out termination point messages to all relevant child sketchlets. In our example, sketchlet C sends a termination point control message to D upon processing the stream packet corresponding to the termination point sent by sketchlet A. After the entire subtree has seen all messages up to the termination point, they acknowledge the initiator sketchlet and start transferring their states asynchronously. Once the parent sketchlet receives acknowledgments from the entire subtree, it propagates the *protocol end* messages to release locks. Locks are released from the bottom to top in the subtree, with the parent sketchlet releasing its lock after each child has released its lock.

3.5 Query Evaluations

Synopsis incorporates support for user-defined queries that are evaluated over the distributed sketch. Queries can be specified by the user in a SQL-like format or with JSON-based key-value descriptions similar to GraphQL [19]. Exact-match, range-based, and summarization queries are all supported over spatiotemporal bounds and individual feature values. The following example depicts how SQL-like queries can be formulated for evaluation over the sketch.

```
SELECT MEAN(precipitation), MAX(wind_speed)
WHERE temperature > 20 AND temperature < 30
AND humidity > .8 AND CORRELATION(
  cloud_cover, precipitation) < -0.25
```

Depending on scaling operations and the spatial scope of the queries, evaluations are carried out on one or more sketchlets. Information on the placement of sketchlets in the system and their corresponding feature scopes is maintained at each sketchlet in a geohash prefix tree, with changes propagated through the network in an eventually-consistent manner as data is ingested and scaling maneuvers occur.

The entry point for these queries, called the *conduit*, may be any of the sketchlets comprising the distributed sketch. During query evaluations, the first step is to identify the set of sketchlets relevant to the query. The conduit consults its prefix tree to locate sketchlets based on spatial, chronological, and feature constraints specified by the user. For spatial constraints that overlap multiple geohashes, a set of *minimum bounding geohashes* (MBG) is constructed based on query geometry. Points falling outside the MBG are trimmed at the individual sketchlet level, in a fashion similar to traversing an R-Tree [20]. After this process is complete, the conduit forwards the queries on to the sketchlets for evaluation and supplies the client with a list of responding sketchlets. As queries execute, results are streamed back and merged by the client API. This strategy ensures that I/O and processing activities are interleaved.

In some situations, a trade-off arises between false positives and false negatives when queries overlap but do not completely cover quantization bins. Users are made aware of this through the error bounds provided by Synopsis, but in cases where false positives are undesirable, partially-matching bins can be pruned from the results. This also affects spatial queries, albeit much less often; while high-resolution geohashes are stored in the SIFT, the encoding scheme is inherently lossy and may produce false positives depending on the configured resolution.

Our distributed prefix tree enables query evaluations during both scaling in and out. For instance, when a conduit attempts to forward a query to a child sketchlet that is undergoing a scale-in operation, the request will be redirected to the its parent sketchlet. This process can continue recursively up through the network, ensuring queries will reach their destination.

3.5.1 Query Types Supported by Synopsis

Relational Queries describe the feature space in the context of the hierarchical trees within our SIFT data structure and may target ranges of values; e.g., "What is the relationship between temperature and humidity during July in Alaska, when precipitation was greater than 1 cm?" These queries return a subset of the overall sketch.

Statistical Queries allow users to explore statistical properties of the observational space. For example, users can retrieve and contrast correlations between any two features at different geographic locations at the same time. Alternatively, queries may contrast correlations between different features at different time ranges at the same geographic location. These queries also

TABLE 2
Local Query Evaluation Times (1,000 Iterations)

Query Type	Mean (ms)	Std. Dev. (ms)
Density	0.007	0.005
Set Cardinality	0.154	0.088
Set Frequency	0.036	0.019
Set Membership	0.015	0.009
Statistical	0.002	0.001
Tree Only (5 km)	46.357	1.287
Tree + Meta (5 km)	40.510	6.937
Tree + Meta (25 km)	47.619	6.355
Tree + Meta (800 km)	53.620	6.818

support retrieval of the mean, standard deviation, and feature outliers based on Chebyshev's inequality [21].

Density Queries support analysis of the distribution of values associated with a feature over a particular spatiotemporal scope. These include kernel density estimations, estimating the probability of observing a particular, and determining the deciles and quartiles for the observed feature.

Set Queries target identification of whether a particular combination of feature values was observed, estimating the cardinality of the dataset, and identifying the frequencies of the observations. Each type of set query requires a particular data structure, with instances created across configurable time bounds (for instance, every day). Set membership is determined using space-efficient bloom filters [22], while cardinality queries are supported by the HyperLogLog [23] algorithm.

Inferential Queries enable spatiotemporal forecasts to be produced for a particular feature (or set of features). Discrete inferential queries leverage existing information in the distributed sketch to make predictions; aggregate metadata stored in the leaves of the tree can produce two-dimensional regression models that forecast new outcomes across each feature type when an independent variable of interest changes.

Synthetic Data Queries allow users generate representative datasets based on the distributions stored in the sketch and then stream them to client applications for analytics. The size of the dataset may also be specified; for instance, 10 percent of the volume of the original data points.

Table 2 outlines tree traversal times for query evaluations. These queries were separated into two groups: conventional lookups and tree retrievals. Conventional lookups include density queries, set queries, and statistical queries, while tree retrievals request targeted portions of the SIFT. While conventional lookups do not return a tree structure to the client, they still require a tree traversal to resolve. In general, tree retrievals consume more processing time due to their serialization and I/O costs; however, it is worth noting that varying the geographical scope across sketchlet sizes from 5 to 800 km did not result in a proportionate increase in processing time.

3.6 Coping with Failures in Synopsis

Synopsis relies on passive replication to recover from sketchlet failures because active replication increases resource consumption significantly and it is infeasible to use upstream backups because the state of a sketchlet depends on the entire set of messages it has processed previously [24].

Support for fault tolerance is implemented by augmenting the distributed sketch with a set of secondary sketchlets. A sketchlet is assigned a set of n secondary sketchlets on different machines to ensure Synopsis can withstand up to n concurrent machine failures. In our implementation, we used two secondary sketchlets ($n = 2$) assigned to each sketchlet. A primary sketchlet periodically sends the changes to its in-memory state as an *edit stream* to its secondary sketchlets. The secondary sketchlets, which act as the sink to the edit stream, serialize incoming messages to persistent storage. This incremental checkpointing scheme consumes less bandwidth compared to a periodic checkpointing scheme that replicates the entire state [24]. By default, Synopsis uses the disk of the machine executing the secondary sketchlet as the persistent storage, but highly-available storage implementations such as HDFS [7] can be used if necessary. To reduce resource footprints, secondary sketchlets do not load the serialized state into memory unless they are promoted to being a primary.

System-wide incremental checkpoints are orchestrated by a special control message emitted by the stream ingesters. Synopsis uses upstream backups at stream ingesters to keep a copy of the messages that entered the system since the last successful checkpoint. In case of a failure, all messages since the last checkpoint will be replayed. Sketchlets are implemented as idempotent data structures using message sequence numbers, hence they will process a replayed message only if it was not processed before. The interval between incremental checkpoints can be configured based on time or the number of emitted messages. Frequent checkpoints can incur high overhead, whereas longer periods between successive checkpoints consume more resources for upstream backups and require longer replay durations in case of a failure.

Membership management is implemented using Zookeeper [25], which is leveraged to detect failed sketchlets. Upon receiving notification of a primary sketchlet failure, a secondary sketchlet assumes the role of primary through a leader election algorithm. The secondary will start processing messages immediately and begins populating its state from persistent storage in the background. Given this mode of operation, there may be a small window of time during which the correctness of queries are impacted. This is rectified once the stored state is loaded to memory and the replay of the upstream backup is completed. The SIFT's support for merge operations as well as its ability to correctly process out of order messages is useful during the failure recovery process.

As per our fault tolerance scheme, the *total time to recover from the failure* (T_{total}) can be modeled as follows:

$$T_{total} = T_d + \max(T_l, T_r),$$

where T_d = *time to detect a failure*, T_l = *time to load persisted state* and T_r = *replay time for messages at the upstream node*.

The time required to detect failures mainly depends on the session timeout value used by Zookeeper to detect lost members and the delay in propagating the notification to other members. With a $5s$ session timeout in an active cluster, we observed a mean notification propagation delay of 5.5 s (std. dev. = 0.911 s, 95th percentile = 6.000 s). Configuring a lower session timeout will increase the chance of false positives if sketchlets become non responsive for a while due to increased load or system activities such as garbage collection. The time required to load the persisted storage depends on the size of the serialized sketchlet; we benchmarked the time it takes to repopulate the state in all sketchlets after ingesting NOAA data for 2014. The mean state re-population time was recorded as 16.602 s with std. dev. = 23.215 s and 95th %ile = 70.877 s. Replay time mainly depends on the

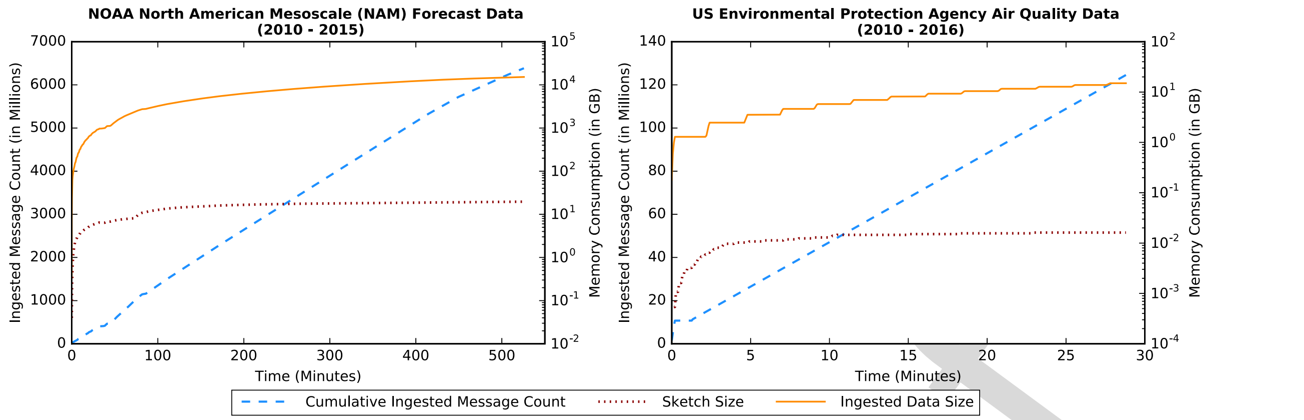


Fig. 8. Memory usage of the distributed sketch over time against the amount of ingested data. The rate of growth decreases over time due to the compact nature of sketchlet data structure.

checkpointing interval as well as the message ingestion rate. With a checkpointing interval of 10000 messages, we experienced a mean value of 0.662 s (std. dev. = 0.026 s, 95th %ile = 0.707 s) to replay an upstream buffer.

4 Performance Evaluation

Here we report system benchmarks profiling several aspects of Synopsis, including the memory consumption and data ingestion performance of the sketch, its ability to handle variable loads, organization of sketchlets, and query performance.

4.1 Dataset and Experimental Setup

We used two datasets for our evaluation. The first was sourced from the NOAA North American Mesoscale (NAM) Forecast System [12]. The NAM collects atmospheric data several times per day and includes features of interest such as surface temperature, visibility, relative humidity, snow, and precipitation. The size of this entire source dataset was 25 TB. The other dataset, collected by the US Environmental Protection Agency, contained daily summary data of four criteria gases (O_3, SO_2, CO and NO_2) used for calculating the air quality in a given area [26]. Each observation in both datasets also incorporates a relevant geographical location and time of creation. This information is used during the data ingest process to partition streams across available sketchlets and preserve temporal ordering of events. Data streams were ingested at faster rates to simulate high data arrival rates while ensuring temporal ordering was preserved.

Performance evaluations reported here were carried out on a cluster of 40 HP DL160 servers (Xeon E5620, 12 GB RAM). The test cluster was configured to run Fedora 24, and Synopsis was executed under the OpenJDK Java runtime 1.8.0_72. For evaluations involving Apache Spark, we used Apache Spark version 2.0.1 with HDFS 2.6.0 with a 100 node cluster by combining our baseline cluster of 40 machines with 30 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM) and 30 HP DL60 servers (Xeon E5-2620, 16 GB RAM).

4.2 Distributed Sketch Memory Evaluation

We monitored the growth in memory consumption of the entire distributed sketch over time with continuous data ingestion as shown in Fig. 8 for both datasets. As more data was streamed into the system, the growth rate decreased as the sketchlets expanded to include vertices for their particular feature space. At the end of our monitoring period, the total amount of ingested data was around three orders of magnitude higher ($\sim$ 1285 for NOAA data and $\sim$ 926 for air quality data) than the in-memory sketch size, resulting in notable space savings.

4.3 Sketch Ingestion Rate

In this experiment, we assessed the ability of the sketch to keep pace with the high rates of incoming observational streams. We partitioned our dataset based on timestamps of observations such that each partition comprised observations for a contiguous time period. Within a partition, data collected in a single observation cycle for all geographical locations were stored as successive records. Records within a single observation cycle were stored in the same order based on their locations across all observational cycles in all partitions. Each partition was assigned a single ingester that sequentially parsed and streamed these records to the distributed sketch. This organization of observations ensured that multiple stream ingesters target a small subset of the sketchlets to profile the *worst case* performance under high stress. This setup forces the corresponding SIFT trees to fan out on different planes (time and features) simultaneously, representing a strenuous workload for the sketch. A real world scenario is simulated with a single partition.

Table 3 summarizes the results of this benchmark. As we increase the number of ingesters with a single sketchlet, the throughput decreases due to the simultaneous fan-out operations taking place within the SIFT trees. This claim is further supported by the increase in the latency for updating the sketchlet as shown in the table. We started with a single sketchlet, allowed the system to dynamically scale out, and measured its throughput once a steady state was reached (i.e., frequent scaling does not occur). The system reached stability with 14-16 sketchlets depending on the number of ingesters. We observed higher throughput compared a single sketchlet due to parallel processing of the observational stream, but the increase was not linear; when there is a single ingester, throughput is constrained by the bandwidth of the ingester. In this benchmark, Synopsis was using around 86 percent of the available bandwidth. With multiple ingesters, due to the way the stream is (intentionally) constructed, the load is not evenly partitioned across the cluster.

4.4 Analyzing a Snapshot of the Distributed Sketch

Fig. 9 visualizes a snapshot of the distributed sketch which demonstrates the organization of sketchlets at runtime as

TABLE 3
Profiling the Update Performance of Sketchlet and Sketch at High Data Ingest Rates

Ingester Count	Sketchlet Throughput (msgs/s)		Sketch Throughput (msgs/s)		Sketchlet Update Latency (μs)		
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	95th Perc.	Std. Dev.
1	15,124.562	575.728	44,082.476	5,984.503	64.752	67.175	5.503
2	14,067.452	491.783	44,060.889	6,206.208	64.971	71.170	4.012
4	11,319.321	1,003.462	41,645.317	13,553.462	74.026	78.364	3.125
8	5,223.280	717.254	38,369.745	14,008.308	81.034	85.842	2.502

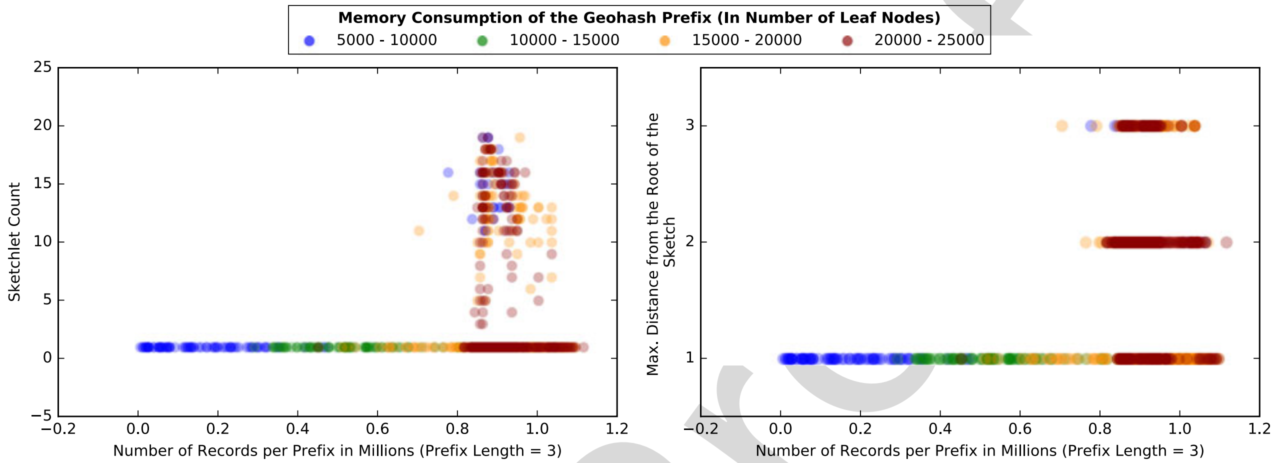


Fig. 9. Analysis of a snapshot of the distributed sketch during data ingestion demonstrating the size and distribution of the information corresponding to different prefixes against the observed record count. If the information is dispersed over multiple sketchlets, it is likely to be a prefix with higher number of records and/or a wide range of observed values.

described in Section 3. This represents the state of the system after consuming the complete 2014 NOAA dataset, resulting in 48 sketchlets. The figure shows the distribution and size of the information maintained across sketchlets for each geohash prefix of 3 characters against the number of records processed for that particular prefix. The memory requirement for a particular geohash prefix depends on the number of records as well as the range of the observed values for different features. The space requirement is measured by the number of leaf nodes in the corresponding sketchlets. For the majority of the prefixes, the space requirement increases with the number of records processed. If the data for a particular prefix is distributed across multiple sketchlets, then it is more likely to be a prefix with a high number of records as shown in the first subplot. In such cases, some of these sketchlets are created in multiple scale-out iterations, which results in a higher distance from the root of the prefix tree. This is depicted in the second subfigure of Fig. 9. A few prefixes with a high number of records can be observed with low memory consumption, and are distributed across multiple sketchlets; their observations span a smaller range, hence they require less memory but were chosen for scaling out operations due to their high message rates.

4.5 Dynamic Scaling: Responding to Variable Load

We evaluated how Synopsis dynamically scales when the data ingestion rate is varied. The data ingestion rate was varied over time such that the peak data ingestion rate is higher than the highest possible cumulative throughput to create a backlog at sketchlets. We augmented the sketch update code with additional operations to match the relatively low ingestion rates used for better control. We used the number of sketchlets within the system to quantify the scaling activities. If the system scales out, more sketchlets will be created as a result of targeted load migration. We started with a single sketchlet and allowed the system to dynamically scale. As can be observed in Fig. 10, the number of sketchlets varies with the ingestion rate. Since we allow aggressive scale-out, rapid scaling out is observed during high data ingestion rates whereas scaling in takes place gradually with one subregion (one sketchlet) at a time.

4.6 Query Evaluation Performance

To evaluate distributed query performance, we executed representative workloads based on observed access patterns over our test dataset across a variety of sketchlet sizes. These queries were categorized as conventional lookups and tree retrievals. Fig. 11 depicts the end-to-end efficiency of the query evaluations over the distributed sketch. Cumulative query throughput and latencies were measured with varying numbers of *concurrent query funnels*. A query funnel continuously generates and dispatches representative queries at the maximum possible rate to stress test the system and saturate its capacity. For example, a query could request summary statistics or feature relationships when the temperature is 20–30 degree, humidity is above 80 percent, and the wind speed is 16 km/h. These queries fluctuated in both the ranges of values and spatial scope, resulting in high variability in the number of sketchlets required to resolve the query as well as the depth and breadth of the tree traversals.

Next we evaluated the query speedup gained by maintaining an in-memory sketch of the data compared to a traditional ETL pipeline. We extracted the timestamp, location information (as a geohash), temperature, surface visibility, humidity

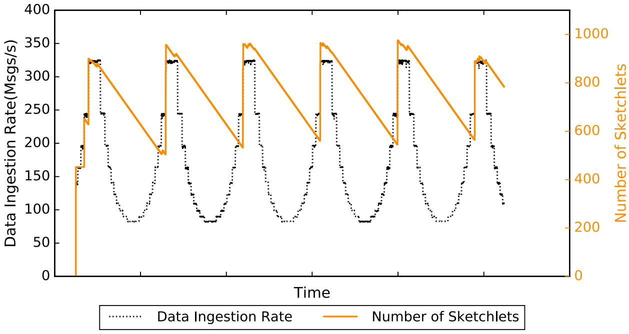


Fig. 10. Responding to variable load using dynamic scaling.

and precipitation in the southeast United States during the months of May–August, 2011–2014 and loaded them into Spark as a DataFrame which was then queried using Spark SQL. Given that the underlying RDD of the DataFrame cannot be shared between multiple Spark applications, we used a multi-threaded driver to issue concurrent queries. Similarly, SYNOPSIS was evaluated using a multi-threaded query funnel. In order to minimize the data transfer between the Spark cluster and the driver, a `count` action was performed on the results of the SQL query and its result was retrieved at the client. For SYNOPSIS, we performed equivalent tree retrieval queries where sections of the distributed sketch is serialized and sent back to the query funnel. End-to-end latencies of the queries were recorded for different concurrency levels. Spark was evaluated under two different settings: caching enabled and disabled for the Dataframe. The results of this evaluation is depicted in Fig. 12. When caching is enabled, the Dataframe will be pinned in memory once materialized for the first time reducing the subsequent access times. Caching the entire Dataframe in memory may not be feasible in most real world spatiotemporal analytical tasks where the size of the dataset exceeds the available memory capacity of the cluster. The end-to-end latency of the SYNOPSIS queries is significantly less despite the larger size of the query results (section of the sketch for SYNOPSIS vs the number of matching records for Spark SQL) transferred from the cluster to the query funnel. Spark queries provides higher accuracy because queries are answered after scanning the entire dataset, but it requires more resources—mainly memory and storage—and incurs higher query latencies. Resource requirements and query latencies with such ETL systems drastically increase with the number of features and geospatial and temporal scopes.

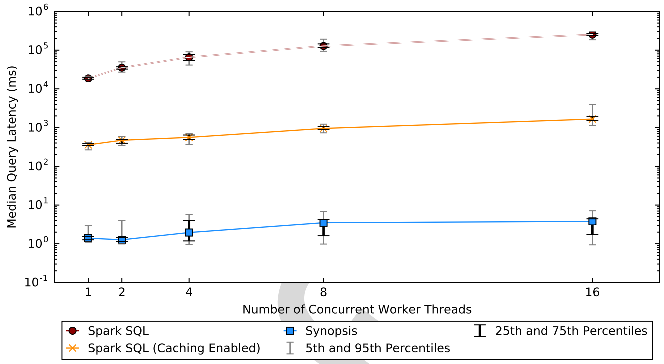


Fig. 12. Contrasting SYNOPSIS query performance with an ETL system built with Spark SQL.

5 APPLICATIONS

Herein we profile the effectiveness of SYNOPSIS as a surrogate for on-disk data in visualization and analytical settings.

5.1 Visualization

To demonstrate the potential applications of SYNOPSIS, we created two visualizations. Our first visualization generated a climate chart by issuing statistical queries to retrieve high, low, and mean temperature values as well as precipitation information for a given spatial region. Climate charts are often used to provide a quick overview of the weather for a location; Fig. 13 summarizes the temperature and precipitation in Snowmass Village, Colorado during 2014. While a standard approach for producing these visualizations over voluminous atmospheric data would likely involve several MapReduce computations, our sketchlets make all the necessary information readily available through queries, avoiding distributed computations altogether. Furthermore, retrieving the data for this evaluation consumed considerably less time (1.5 ms) than rendering the image on the client side (127.1 ms).

Our second visualization issued queries to retrieve cloud cover information for the entirety of North America. To reduce processing load on the client side, we specified minimum visibility thresholds to eliminate data points that would not be visible in the final output figure. After retrieving this information, we executed a second query that located all areas that exhibited high correlations between cloud cover and precipitation. Fig. 14 illustrates the results of this process for data in July of 2014; cloud cover is

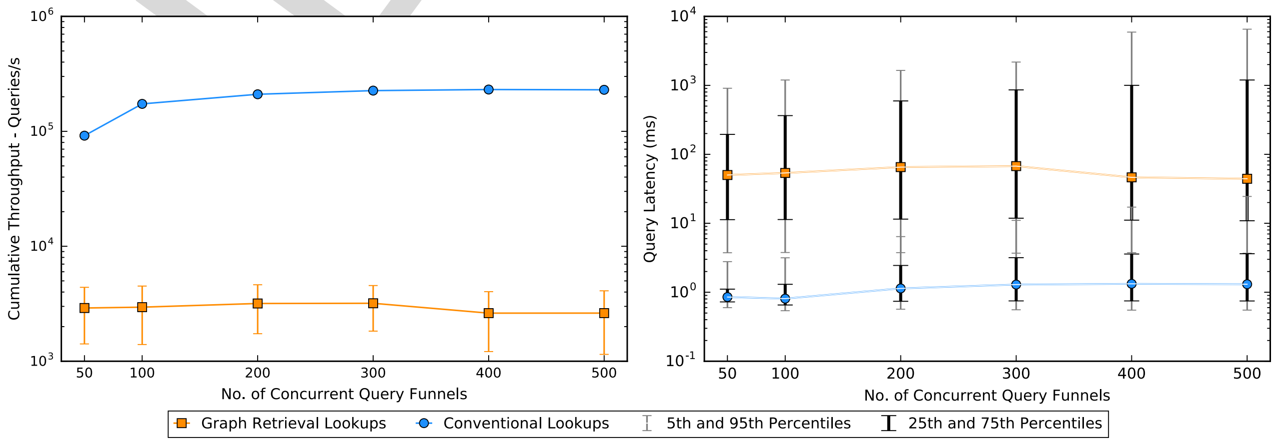


Fig. 11. Distributed query evaluation performance — cumulative throughput and latency in a 40-node SYNOPSIS cluster.

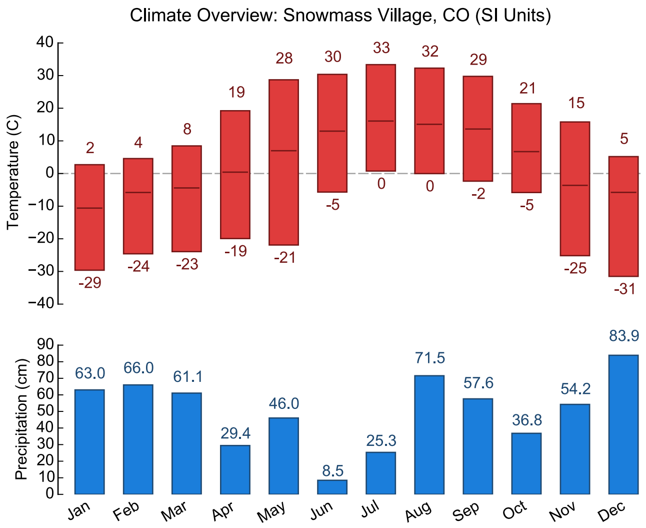


Fig. 13. A climate chart generated using a statistical query.

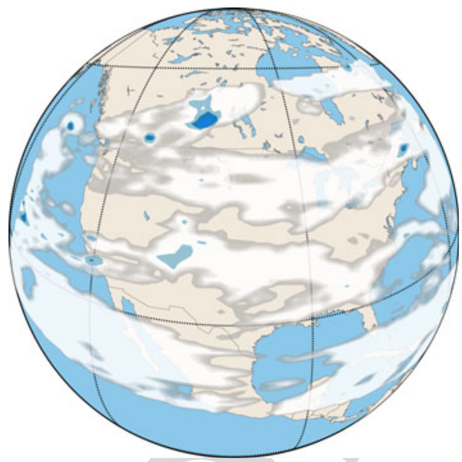


Fig. 14. Global contour visualization showing cloud cover (white contours) and the correlation with precipitation (blue contours) in July of 2014 across North America.

represented by white contours with varying opacity, while blue contours describe the correlation between cloud cover and precipitation (darker blues, such as those seen in the top-center of the globe, represent a stronger correlation). Due to the large scope of this visualization (retrieving all data points for a given month across all spatial regions), retrieval took approximately 2.82 seconds, with graphics rendering consuming an additional 1.51 seconds at the client application.

5.2 Use with Analytic Engines

Synthetic data queries in SYNOPSIS can be used to generate representative datasets that require less space while still providing high accuracy. Such datasets can be used efficiently with analytic engines such as Apache Spark [1] and TensorFlow [4]. We used Spark to train regression models based on the Random Forest ensemble method to predict temperatures (in Kelvin) using surface visibility, humidity and precipitation in the southeast United States during the months of May–August, 2011–2014. These models were generated using the original full-resolution data as well as synthetic data sets that were sized at 10, 20, and 100 percent of the original data. For another point of comparison, we also generated datasets using 10 and 20 percent samples of the original data. The accuracy of these models was measured using a test dataset extracted from actual observations (30 percent of the overall dataset size). All five datasets were staged on HDFS and loaded into Spark to train the models.

We evaluated our approach based on the on-disk and in-memory storage requirements, data loading time, training time and the accuracy of the model. Our observations are summarized in Table 4; overall, the accuracy of the synthetic data models is comparable to that of the actual data, while requiring less space, training, and loading times; for instance, our 10 percent synthetic dataset produces a model with similar accuracy while incurring 54 percent less training time and reducing space requirements by 90 percent. Additionally, based on the number of RDD partitions used, the smaller synthetic datasets require substantially less computing resources. It is worth noting that these particular models do not appear to benefit from a larger set of training samples, and could potentially begin to exhibit over-fitting if trained on more data. We believe signs of this are demonstrated by the 100 percent synthetic dataset, where fidelity limits of the sketch result in training data points that slightly decrease the expressiveness of the model.

6 RELATED WORK

Tao et al. [27] answers distinct sum and count queries over spatiotemporal data with a sketch index similar to an aRB-tree [28] where spatial indexing is implemented is using an R-tree and temporal indexing is implemented as a B-tree. At the leaf nodes of the B-tree, a sketch that follows the Flajolet-Martin algorithm [29] is used to capture an approximate view of the observations. This approach significantly reduces the space requirements for answering distinct sum/count queries on spatiotemporal data and provides efficient query evaluations due to its ability to prune portions of the search space. SYNOPSIS differs in its ability to capture multiple features and their interactions, which facilitates a broader set of queries.

TABLE 4
Comparing Random Forest Based Regression Models Generated by Spark MLlib Using Synthetic Versus Real Data

Dataset	Size (GB)	RDD Partitions	Data Loading Time (s)		Model Training Time (s)		Accuracy - RMSE (K)	
			Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Original	25.350	208	28.035	2.249	506.493	9.500	5.981	0.027
Original - 10% Sample	2.535	21	5.136	0.909	205.627	5.798	5.960	0.049
Original - 20% Sample	5.069	41	6.102	1.607	216.857	7.994	5.994	0.026
Synthetic - 10%	2.549	21	5.097	0.912	231.221	13.459	5.951	0.027
Synthetic - 20%	5.098	41	6.208	0.637	235.018	16.148	5.981	0.051
Synthetic - 100%	25.066	207	25.670	2.947	454.964	16.446	6.192	0.076

Data Cubes [30], [31], [32], [33] are a data structure for Online Analytical Processing that provide multidimensional query and summarization functionality. These structures generalize several operators provided by relational databases by projecting two-dimensional relational tables to N-dimensional cubes (also known as *hypercubes* when $N > 3$). Variable resolution in Data Cubes is managed by the *drill down/drill up* operators, and *slices* or entire cubes can be summarized through the *roll up* operator. While Data Cubes provide many of the same features supported by SYNOPSIS, they are primarily intended for single-host offline or batch processing systems due to their compute- and data-intensive updates. In fact, many production deployments separate transaction processing and analytical processing systems, with updates pushed to the Data Cubes periodically.

Galileo [34], [35] is a distributed hash table that supports the storage and retrieval of multidimensional data. Given the overlap in problem domain, Galileo is faced with several of the same challenges as SYNOPSIS. However, the avenues for overcoming these issues diverge significantly due to differences in storage philosophy: SYNOPSIS maintains its dataset completely in main memory, avoiding the orders-of-magnitude disparity in I/O throughput associated with secondary storage systems. This makes SYNOPSIS highly agile, allowing on-demand scaling to rapidly respond to changes in incoming load. Additionally, this constraint influenced the trade-off space involved when designing our algorithms, making careful and efficient memory management a priority while striving for high accuracy.

Simba (Spatial In-Memory Big data Analytics) [36] extends Spark SQL [2] to support spatial operations in SQL as well as DataFrames. It relies on data being stored in Spark [1]. Despite its higher accuracy, it is not scalable for geospatial streams in the long term due to high storage requirements. In SYNOPSIS, spatial queries can be executed with a reasonable accuracy without having to store the streaming data as-is.

Dynamic scaling and elasticity in stream processing systems has been studied thoroughly [24], [37], [38], [39], [40], [41]. StreamCloud [38] relies on a global threshold-based scheme to implement elasticity where a query is partitioned into sub-queries which run on separate clusters. It relies on a centralized component, the Elastic Manager, to initiate the elastic reconfiguration protocol, whereas in SYNOPSIS each node independently initiates the dynamic scaling protocol. This difference is mainly due to different optimization objectives of the two systems; StreamCloud tries to optimize the average CPU usage per cluster while SYNOPSIS attempts to maintain stability at each node. The state recreation protocol of StreamCloud is conceptually similar to our state transfer protocol, except that tuples are buffered at the new location until the state transfer is complete, whereas in SYNOPSIS the new sketchlet immediately starts building the state which is later merged with the state (transferred asynchronously) from the parent sketchlet.

Gedik et al. [41] also uses a threshold-based local scheme similar to SYNOPSIS. Additionally, this approach keeps track of the past performance achieved at different operating conditions in order to avoid oscillations in scaling activities. The use of consistent hashing at the splitters (similar to geohash based stream partitioning in SYNOPSIS) achieves both load balancing and monotonicity (elastic scaling does not move states between nodes that are present before and after the scaling activity). Similarly, our geohash-based partitioner together with control algorithms in SYNOPSIS balance the workload by alleviating hotspots and sketchlets with lower resource utilization. Our state migration scheme doesn't require migrating states between sketchlets that do not participate in the scaling activity, unlike with a reconfiguration of a regular hash-based partitioner. Unlike in SYNOPSIS, in their implementation, the stream data flow is paused until state migration is complete using vertical and horizontal barriers. Finally, SYNOPSIS' scaling schemes are placement-aware, meaning certain nodes are preferred when performing scaling with the objective of reducing the span of the distributed sketch.

7 CONCLUSIONS AND FUTURE WORK

SYNOPSIS, our framework for constructing a distributed sketch over spatiotemporal streams, is able to (1) maintain a compact representation of the observational space, (2) support dynamic scaling to preserve responsiveness and avoid overprovisioning, and (3) explore the observational space with a rich set of queries. Our methodology for achieving this is broadly applicable to other stream processing systems and our empirical benchmarks demonstrate the suitability of our approach.

We achieve compactness in our sketchlet instances by dynamically managing the number of vertices in the SIFT hierarchy as well as the range each vertex is responsible for. We also maintain summary statistics and metadata within these vertices to track the distribution/dispersion of feature values and their frequencies. As a result, SYNOPSIS is able to represent datasets using substantially less memory (*RQ-1*). Given variability in the rates and volumes of data arrivals from different geolocations, our scaling mechanism avoids overprovisioning and alleviates situations where sketch updates cannot keep pace with data arrival rates. Memory pressure is also taken into account during replica creation as well as when scaling in and out (*RQ-2*). During evaluations, only the sketchlets that hold portions of the observational space implicitly or explicitly targeted by the query are involved, ensuring high throughput. We support several high-level query operations allowing users to locate and manipulate data efficiently (*RQ-3*).

Our future work will target support for SYNOPSIS to be used as input for long-running computations. Such jobs would execute periodically on a varying number of machines and could target the entire observational space or only the most recently-assimilated records. We also plan to implement continuous queries that can autonomously evolve with the feature space.

ACKNOWLEDGMENTS

This research has been supported by funding from the US Deptartment of Homeland Security [D15PC00279]; the US National Science Foundation [ACI-1553685, CNS-1253908]; and a Monfort Professorship.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, pp. 10–10, 2010.
[2] M. Armbrust, et al., "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[3] M. Abadi, "Tensorflow: Learning functions at scale," in *Proc. 21st ACM SIGPLAN Int. Conf. Functional Program.*, 2016, pp. 1–1.

[4] M. Abadi, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[5] ASF, "Hadoop," 2016. [Online]. Available: http://hadoop.apache.org

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, May 2010, pp. 1–10.

[7] D. Borthakur, "Hdfs architecture guide," *Hadoop Project*, 2008.

[8] J. Langford, et al., "Vowpal wabbit online learning project," 2007.

[9] T. Buddhika and S. Pallickara, "NEPTUNE: Real time stream processing for internet of things and sensing environments," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 1143–1152.

[10] T. Buddhika and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS.2017.2723403.

[11] G. Niemeyer, Geohash, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[12] National Oceanic and Atmospheric Administration, The North American Mesoscale Forecast System, 2016. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[13] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[14] M. Kristan, A. Leonardis, and D. Skocaj, "Multivariate online kernel density estimation with gaussian kernels," *Pattern Recognit.*, vol. 44, no. 10/11, pp. 2630–2642, 2011.

[15] M. Malensek, et al., "Autonomously improving query evaluations over multidimensional data in distributed hash tables," in *Proc. ACM Cloud Autonomic Comput. Conf.*, Sep. 2013, pp. 15:1–15:10.

[16] B. W. Silverman, *Density estimation for statistics and data analysis*. CRC press, 1986, vol. 26.

[17] M. Maurer, S. Pallickara, and S. Pallickara, "Revealing the MAPE loop for the autonomic management of cloud infrastructures," in *Proc. IEEE Symp. Comput. Commun.*, 2011, pp. 147–152.

[18] T. Lorido-Botr án, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," Dept. Comput. Archit. Technol., Univ. Basque Country, Vizcaya, Spain, Tech. Rep. EHU-KAT-IK-09, vol. 12, 2012.

[19] Facebook, Inc, GraphQL, 2015. [Online]. Available: graphql.org

[20] A. Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching*. New York, NY, USA: ACM, 1984.

[21] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms (Volume 1)*," Reading, MA, USA: Addison-Wesley, 1968.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Int. Conf. Anal. Algorithms*, 2007, pp. 127–146.

[24] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.

[25] P. Hunt, et al., "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Techn. Conf.*, vol. 8, 2010, Art. no. 9.

[26] US Environmental Protection Agency, Daily summary data - Criteria gases, 2017. [Online]. Available: https://aqsdr1.epa.gov/aqsweb/aqstmp/airdata/download_files.html#Daily

[27] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias, "Spatio-temporal aggregation using sketches," in *Proc. Int. Conf. Data Eng.*, Mar. 2004, pp. 214–225.

[28] D. Papadias, et al., "Indexing spatio-temporal data warehouses," in *Proc. Int. Conf. Data Eng.*, 2002, pp. 166–175.

[29] P. Flajolet and G. Nigel Martin , "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[30] J. Gray, et al., "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," in *Proc. 12th Int. Conf. Data Eng.*, 1996, pp. 152–159.

[31] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1996, pp. 205–216.

[32] I. S. Mumick, D. Quass, and B. Singh Mumick , "Maintenance of data cubes and summary tables in a warehouse," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, pp. 100–111.

[33] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range queries in olap data cubes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1997, pp. 73–88.

[34] M. Malensek, S. Pallickara, and S. Pallickara, "Analytic queries over geospatial time-series data using distributed hash tables," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 6, pp. 1408–1422, Jun. 2016.

[35] M. Malensek, S. Pallickara, and S. Pallickara, "Fast, ad hoc query evaluations over multidimensional geospatial datasets," *IEEE Trans. Cloud Comput.*, vol. 5, no. 1, pp. 28–42, Jan. 2017.

[36] D. Xie, et al., "Simba: Efficient in-memory spatial analytics."

[37] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. IEEE 30th Int. Conf. Data Eng. Workshops*, 2014, pp. 296–302.

[38] V. Gulisano, et al., "Streamcloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

[39] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An elastic and highly available streaming service in the cloud," in *Proc. Joint EDBT/ICDT Workshops*, 2012, pp. 55–60.

[40] T. Heinze, et al., "Elastic complex event processing under varying query load," in *Proc. 1st Int. Workshop Big Dynamic Distrib. Data*, 2013, pp. 25–30.

[41] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proc. IEEE Int. Symp. Parallel Distrib. Processing*, 2009, pp. 1–12.

Thilina Buddhika is working toward the PhD degree in the Computer Science Department, Colorado State University. His research interests include area of real time, high throughput stream processing specifically targeted to environments such as Internet of Things (IoT) and health care applications.

Matthew Malensek received the PhD degree from the Department of Computer Science, Colorado State University. His research involves the design and implementation of large-scale distributed systems, data-intensive computing, and cloud computing.

Sangmi Lee Pallickara received the master's and PhD degrees in computer science from Syracuse University and Florida State University, respectively. She is an associate professor in the Department of Computer Science, Colorado State University. Her research interests are in the area of large-scale scientific data management. She is a recipient of the NSF CAREER award.

Shrideep Pallickara received the master's and PhD degrees from Syracuse University. He is an associate professor in the Department of Computer Science and a Monfort Professor, Colorado State University. His research interests include large-scale distributed systems. He is a recipient of an NSF CAREER award. He is member of the IEEE.

Queries to the Author

Q1. When you submit your corrections, please either annotate the IEEE Proof PDF or send a list of corrections. Do not send new source files as we do not reconvert them at this production stage.

Q2. There was a discrepancy in Bibliography in the PDF and the source file. We have followed the source file.

Q3. Please provide complete bibliographic information in Refs. [7], [8], [10], and [36].

Q4. You will be charged $200 in MOPC fees because your paper is 15 pages.