Enabling Fast Exploratory Analyses Over Voluminous Spatiotemporal Data Using Analytical Engines

Daniel Rammer, Thilina Buddhika, Matthew Malensek, Shrideep Pallickara, Sangmi Lee Pallickara, *Members, IEEE,*

Abstract—Fueled by the proliferation of IoT devices and increased adoption of sensing environments the collection of spatiotemporal data has exploded in recent years. Disk based storage systems provide reliable archives but are far too slow for efficient analytics. Furthermore, spatiotemporal datasets quickly exceed the memory capacity of cluster environments. Current solutions focused on in-memory analytics suffer from memory contention and unnecessary network I/O, failing to provide a suitable platform for iterative, exploratory analytics in shared environments. In this work we propose Anamnesis, the first in-memory, sketch aligned, HDFS compliant storage system. Data sketching algorithms reduce dataset sizes by summarizing feature values and inter-feature relationships. Anamnesis leverages data sketches to alleviate memory contention and vastly reduce network I/O during analytics. Upon request, we generate accurate full-resolution datasets with negligible resource and time costs. Datasets are available using a fully HDFS compliant interface allowing Anamnesis to achieve unprecedented compatibility with popular analytics engines. This facilitates adoption into existing workflows by serving as a "drop-in" replacement for canonical HDFS. We evaluate the system using 2 spatiotemporal datasets, a variety of popular analytics engines, and real-world analytical operations.

Index Terms—data sketches, spatiotemporal data, big data, distributed analytics

1 Introduction

Spatiotemporal data occur in domains exploring natural phenomena, such as atmospheric science, environmental and ecological monitoring, geosciences, and epidemiology, and in commercial and social networking settings.

Spatiotemporal data volumes have grown exponentially over the past decade alongside methodological innovations in data organization, data fitting algorithms, and analytics. New equipment characteristics have exacerbated this growth. In particular, this includes improved hardware capabilities of *in situ* and remote sensing equipment that report measurements with increased precision, improved resolution, and at higher rates. Falling equipment costs also contribute to the proliferation of these devices.

These data offer unprecedented opportunity for exploration and gleaning insights. While efforts have been made to organize and perform timely analytics on voluminous datasets, effective data analysis is stymied by data volumes, disk I/O, and network movement costs.

This study aims to support fast analytics over voluminous, spatiotemporal data.

1.1 Challenges

Challenges in enabling exploratory analytics over voluminous spatiotemporal data stem from data volumes, systems aspects, and the nature of the analytic process.

- *Data volumes:* The datasets we consider are voluminous, encompassing a very large number of multidimensional observations.
- *Sluggish I/O subsystem and the memory hierarchy:* The speed differential of the memory hierarchy compounds these challenges with disk I/O being several orders of magnitude slower than memory. Any data item that is being processed must be memory resident; datasets, or portions thereof, need to be memory resident prior to performing analytic operations. Ultimately, analytic operations are dominated by time spent performing I/O.
- *Choice of analytical engines:* Users prefer to have a choice in their use of analytical engines for data analysis. However, this choice can be difficult to achieve and requires the user to manage several aspects of the data preparation tasks.
- *Performance hotspots:* There may be severe imbalances in how the data are dispersed over a collection of machines. Data movements in these situations can be prohibitive, entailing both disk I/O and network I/O. If data movements to rebalance the data prior to analysis are not performed, the subsequent in-place analysis leads to performance hotspots on machines that hold disproportionate portions of the data space.
- *Data movement costs:* During data staging it is impossible to know the data access patterns triggered by future analytic tasks. For example, data may be staged on disks with the objective of preserving geographical proximity, but the analytics may target temporal analyses across multiple geospatial scopes — such an analytics job may trigger significant data movements.
- *Data formats:* The analysis — esp. legacy code — may expect the data to be in a particular format; this may be different from the format in which the data is stored.
- *Iterative nature of the analytic process:* The analytics process involves identifying portions of the data that are of interest. Once results are obtained the scientist may wish to augment either the analytic operations or further refine the datasets. Data refinements involve updating the sets of features and bounds/thresholds on values that particular features might take. Analytic operations may involve multiple, repeated sweeps of the data (or the portions thereof) — however the data volumes preclude how often these can be performed, if at all.

- *D. Rammer, T. Buddhika, S. Pallickara, and S.L. Pallickara are with the Department of Computer Science, Colorado State University, Fort Collins, CO - 80523. M. Malensek is with the Department of Computer Science at the University of San Francisco, San Francisco, CA - 94117.*

1.2 Research Questions

The following research questions guide our investigations.

RQ-1: *How can we cope with the speed differential of the memory hierarchy?* In particular, we must reduce the amount of disk I/O that needs to be performed during analytic operations.

RQ-2: *How can we minimize data movement costs during analytics over spatiotemporal data?* Since analytic jobs are launched in shared clusters, inefficiencies in network utilization (including contention) affect other users and applications utilizing those clusters.

RQ-3: *How can we support interoperation with diverse analytical engines?* Analytical engines vary in the complexity and variety of operations that they support

RQ-4: *How can we ensure fast completion times to facilitate the iterative analytics process?* Timely completion of analytic operations fosters experimentation that is key to deriving insights.

1.3 Approach Summary

The overarching theme in our methodology is to reduce: memory footprints and contention, data movements, and the amount of disk I/O that needs to be performed. These themes are complemented with a focus on managing the competing pulls of data locality and load balancing during analyses.

In particular, our methodology leverages a distributed, spatiotemporal sketching algorithm (Synopsis [1]) to generate compact, memory-resident sketches of the data space. We use the sketch as a surrogate for the full-resolution data. The sketch is memory-resident and eliminates the need to the write full-resolution, raw observations to disk.

A key step in the analysis process is identifying data that must be analyzed. The in-memory sketch is organized as a prefix tree that straddles multiple machines. Query evaluations over the distributed data sketch result in identification of portions of the data space that are of interest. Tree paths from different portions of the distributed sketch are agglomerated in our *Slice* data structure. Slices encapsulate information as a traversable forest of trees; each tree path represents a spatiotemporal scope for which query constraints were satisfied. We allow users to reorient the trees comprising the forest. These reorientations are aligned with the nature of the analyses being performed. For example, the root node in these trees would be the feature designated as the primary feature for analysis.

Users interact with Slices via two classes of operators: *instrumentation* and *materialization*. Instrumentation operators manage the composition of the dataset. Instrumentation operators can be pairwise or standalone and include set operations, augmenting with auxiliary datasets, and incremental refinements. Set operations that we support include union, intersection, distinct, and subtraction across two or more Slices. Slices can be augmented with auxiliary datasets to supplement the analyses; these result in the addition of new features. A Slice can be incrementally refined to adjust spatiotemporal scopes and constrain the number and type of features that are of interest.

Materialization operators create exploratory datasets that are suitable for analysis using diverse analytical engines. The system manifestation/distribution of the exploratory dataset is aligned with the expected processing with the goal of reducing data movements and the accompanying contention and reduced performance. Materialization results in the creation of an exploratory dataset that comprises data items with values in the units, scale, and formats that the analyses task expects it to be in. A materialized Slice is an exploratory dataset. The materialization class of operators is responsible for ensuring that the data is ready for analyses by diverse analytical engines. In particular, materialization ensures the representativeness of the data space and alleviates the I/O-induced inefficiencies when processing using diverse analytical engines. Our methodology manages the competing pulls of load balancing, data locality, and ensuring fast completion times.

To support interoperation with diverse analytical engines, we materialized exploratory datasets in HDFS (Hadoop Distributed File System [2]). Several analytical engines, such as Hadoop MapReduce, Spark, TensorFlow, Mahout, etc., support integration with HDFS and use it as a primary source for accessing input data. HDFS, which is data-format-neutral and suited for semi/unstructured data, provides an excellent avenue for us to interoperate with analytical engines. Most importantly, users can reuse or modify legacy code that they developed in their preferred analytical engines.

To further enhance the integration with analytical engines we have designed a memory-resident, HDFS-compatible distributed file system, Anamnesis, from the ground up. Since Anamnesis is fully HDFS compliant, its interactions with analytical engines (control-plane traffic) are indistinguishable from the canonical HDFS but augment it with capabilities aligned with our needs and eliminate unnecessary functionality that worsens resource requirements and contention, e.g., replication and fault recovery. Most importantly our methodology eliminates disk I/O when analytical engines access the data as analytic operations are launched.

1.4 Experimental Datasets

We performed benchmarks using two real-world, voluminous datasets. The first contains weather related observations reported by the National Oceanic and Atmospheric Administration (NOAA) and the second is an air quality dataset collected by the US Environmental Protection Agency (EPA). In both datasets we see that consecutively reported observations from a single collection point tend to have relatively small feature changes often noted in time series datasets. However, observations reported between collection points often display greater variability.

The NOAA weather dataset reports observations from 1.3 million globally dispersed collection points. There are 56 floating point decimal features uniformly reported every 6 hours by each collection point. A few example features include: surface temperature, surface wind gust speed, relative humidity, atmospheric precipitable water, and tropopause pressure.

The EPA air quality dataset reports from 275 million U.S. based collection points. Observations are reported every 4 hours and include information on certain gases, particulates, meteorological information, and toxins (including ozone precursors and lead).

Although our experiments focus on environmental datasets, our approach is designed to process data from a variety of domains with a few restrictions. Datasets must be observational, and therefore this work is not applicable to many media applications, including image and video processing, etc. Datasets with spatiotemporal attributes are preferred to effectively leverage advantages provided by the sketching algorithm. It is also important to note that datasets may be static or dynamically reported as the sketch is amendable to incremental additions.

1.5 Paper Contributions

Our methodology supports fast analyses over voluminous, multidimensional, spatiotemporal datasets. In particular our

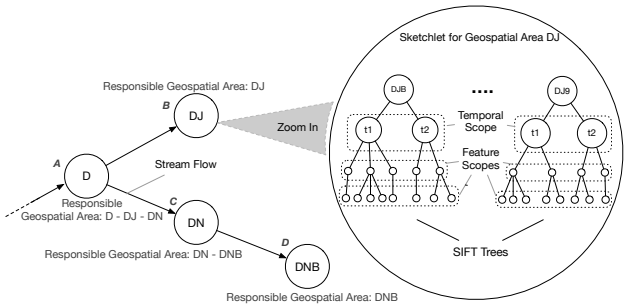


Fig. 1: A demonstration of the distributed sketch for region represented by geohash prefix D. The sketchlets for prefixes DJ and DN have scaled out due to high data densities. Each sketchlet maintains a SIFT, a forest of trees, with each tree responsible for a geospatial subregion.

contributions include:

1) Support for near real-time construction of custom datasets. This includes support for selection of features, selection of the primary feature, and augmenting analysis with auxiliary datasets.
2) Significant reduction in the amount of disk I/O that needs to be performed during analytic operations. This results in faster completion times for analytic operations.
3) Support for performing analyses using diverse analytical engines.
4) Support for fast completion of analytic tasks at high-throughput.
5) To our knowledge, Anamnesis is the first HDFS implementation specifically designed for integration with data sketches.

2 Systems Overview

Herein we present:

1) An overview of the Synopsis sketch for spatiotemporal data;
2) A description of how the components comprising our framework share available physical resources;
3) An overview of the Slice data structure that is the starting point for explorations.

2.1 Synopsis Sketch

Synopsis [1] is a distributed sketch constructed over spatiotemporal data streams. Data must be observational, and the sketch is amenable to incremental updates allowing for dynamically reported datasets. The Synopsis sketch is composed of a number of *sketchlets* dispersed over multiple machines, each responsible for a particular geospatial area. Sketchlets are organized as a prefix tree based on the geohash [3] prefixes corresponding to the geospatial bounding boxes they are responsible for. Synopsis dynamically scales based on the data availability densities at different geospatial regions in the incoming input streams. Scale-out operations perform targeted load migration for regions with higher data densities to relieve memory pressure of the host nodes of the affected sketchlets while preserving the performance of the distributed sketch. Scale-in operations merge sketchlets with low data densities into a single sketch to conserve memory. An example snapshot of the sketch is depicted in Figure 1.

A sketchlet organizes observations corresponding to its region as a group of tree-based representations called *SIFT (Summarization Involving a Forest of Trees)*. The edges and vertices within each SIFT tree maintain interfeature relationships while leaves maintain online summary statistics for individual features and cross-feature covariances using Welford's method [4]. Each level of a SIFT tree represents either the temporal scope or a feature. Compaction is achieved by *grouping* to exploit similarities in values reported within observations instead of storing the raw feature values. Similar feature values are grouped together into bins determined using an online kernel density estimation function. A vertex is created in the SIFT tree for each bin at the appropriate level. Insertion of a multidimensional observation creates a path in the SIFT tree (if the path does not exist) connecting the respective bins for individual feature values. Statistics maintained at the leaf node of the path are also updated.

A Synopsis sketch can be queried using different types of queries, such as relational queries and statistical queries. Any Synopsis node can accept a query and subsequently will disseminate the query to the appropriate machines that are holding the matching sketchlets. Synthetic data generation is a special type of query that allows users to generate representative datasets based on the distributions stored in the sketch.

2.2 System Architecture

The system architecture consists of (1) the Synopsis cluster, (2) Anamnesis cluster, (3) client (driven by the Java API / Scala shell), and (4) analytics engines.

1) **Synopsis Cluster:** Synopsis [1] supports memory efficient data read/write operations. The building block sketchlets (as described in Section 2.1) sketch full-resolution data. The system never writes data to disk; it stores sketches in memory.
2) **Anamnesis:** This is our in-memory HDFS compliant implementation. The application components, namenodes and datanodes, are identical to canonical HDFS in their communication protocols. Therefore, Anamnesis is fully compatible with any system that supports read / write operations through HDFS. The main difference with canonical HDFS is our implementation provides native support for data sketches and effective memory residency while reducing contention.
3) **Java API / Scala Shell:** User interaction with the various components (Synopsis and Anamnesis) is driven by a Java API. The compiled JAR may be imported into any Java compatible application including a dynamic Scala shell. The API provides support for Slice initialization from a Synopsis cluster, configuration / manipulation (instrumentation), and distribution to Anamnesis (materialization).
4) **Analytics Engines:** The system makes data available in an HDFS compliant interface, therefore any analytics engine that supports HDFS may interoperate with our system (e.g., Apache Spark, Hadoop, TensorFlow, etc.).

Interaction between system components is performed in a workflow shown in Figure 2. (1) The client queries the Synopsis cluster using the Java API / Scala shell, organizing data of interest in the locally maintained Slice data structure. (2) The client materializes the Slice over the Anamnesis cluster, distributing data based on proposed analytics. (3) Clients execute analytics operations using popular distributed analytics engines. (4) The Anamnesis cluster dynamically generates synthetic data to serve data requests over an HDFS compliant interface. Each stage is described in Section 3.

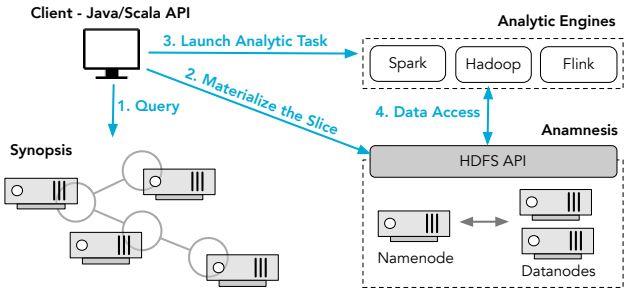


Fig. 2: System architecture displaying interaction of individual components. Scala shell facilitates Slice initialization from Synopsis queries, materialization of Slices over Anamnesis, and execution of analytics tasks using popular distributed analytics engines.

2.3 Slice

A Slice is a data structure that organizes portions of the dataspace that are of interest. Slice organizes information as a traversable forest of trees. We have incorporated capabilities to allow for dynamic manipulation and configuration of datasets. We initialize Slices using results returned from distributed query evaluations on the Synopsis sketch. The Slice allows us to create, refine, and fuse information from sketches of diverse datasets.

3 Methodology

Our methodology to leverage sketches in support of analytics over spatiotemporal data encompasses the following:

1) *The Slice data structure:* The Slice, with its support for data organization and refinements, is the starting point for analytic operations. The Slice plays a key role in organizing portions of the Synopsis sketch that satisfy specified query constraints. [***RQ-1***, ***RQ-2***]
2) *Instrumentation of Slices:* The Slice organizes information identifying the data of interest. We support mechanisms to refine Slices and operations to combine them with other Slices in myriad ways. [***RQ-2***]
3) *Materializing Slices:* A Slice represents a blueprint identifying the dataspace of interest. A Slice must be materialized prior to processing. The materialization encompasses generation of exploratory datasets, grouping of data items within the exploratory datasets as shards to ensure data locality, and load balanced distribution of shards to ensure fast completion times. [***RQ-2***, ***RQ-4***]
4) *Interoperation with analytical engines:* We leverage HDFS as the entry point for interoperating with myriad analytical engines. We also focus on timely completion via two optimization features: memory residency to alleviate I/O costs and just-in-time inflation of shards to reduce the durations for which shards are memory-resident. [***RQ-1***, ***RQ-3***]

3.1 Slice [RQ-1, RQ-2]

A first step in the analytics process is identifying portions of the data space that are of interest. The Synopsis sketch supports relational and statistical query evaluations. The results of these query evaluations identify portions of the sketch that satisfy the specified query constraints. The results manifest as a collection of tree paths from within the Synopsis sketch that are streamed back to the clients from each Synopsis node. Each tree path represents a sliver of the spatiotemporal scope encapsulated by the sketch.

We organize results from these query evaluations in our Slice data structure. The Slice organizes the results as a forest of trees. The Slice supports three organizational features that simplify data space explorations: traversals, agglomerations, and reorientations. A Slice can be programmatically traversed to refine the spatiotemporal scopes or features that are of further interest.

The Slice data structure maintains internal metadata that can be queried. This includes information about the number of tree paths, features, and the underlying spatiotemporal scopes. Information about the number of observations at different spatiotemporal scopes, cross-feature covariances, and range of values for a particular feature can also be retrieved.

Tree paths within a Slice can be agglomerated. For example, tree paths from smaller, contiguous spatial scopes can be combined to produce a smaller set of tree paths under a larger geospatial scope. The leaves of each tree path culminate in Welford statistics that are applicable for the agglomerated tree path. Similar agglomerations can be performed for the temporal dimension.

The Slice may be used to reorient the trees that comprise it. Each reorientation produces a logically equivalent version of the Slice but with differences in the levels associated with the spatial, temporal, and feature-specific components. Reorientations also result in differences between the number of nodes and edges that comprise the Slice; this is because reorientation may change the fan-outs associated with a particular node.

3.1.1 *Java/Scala API*

To facilitate effective system interaction, we imported the Java API into a Scala shell. This offers advantages over a compiled application. Scala provides an interactive shell, enabling dynamic Slice configuration. We provide simple fluent-style interfaces for initializing Slices from Synopsis queries, chaining of instrumentation operators for Slice configuration and refinement, and robust Slice materialization. All of these operations are amenable to moderate updates as iterative analysis requires.

Many commodity analytics engines support Scala integration. For example, Apache Spark provides a Scala shell interface. This integration allows us to easily import our Java API and simultaneously drive Slice definition/distribution and analytics tasks within the same Scala shell. This coupling simplifies iterative analytics.

3.2 Instrumentation of Slices [RQ-2]

The Slice can be refined to precisely identify portions of the data space that are of interest. A user may transform Slices to refine and fuse with other Slices to derive a new Slice that is suitable for analysis. We support two key mechanisms for instrumenting Slices: standalone and pairwise.

In standalone refinements, a Slice is incrementally refined to facilitate precise composition of the dataspace of interest. In particular, a user performs feature selection by including or excluding particular features, and controls the chronological time ranges of the data space and the geographical extents. Feature-specific refinements include selection of the range of values that are of interest; these are useful in cases where a user specifies queries with broad ranges that are then narrowed. Cross-feature refinements can be performed to include/exclude tree paths based on thresholds specified for observed covariances at different spatiotemporal scopes and on the range of values specified for particular feature-value combinations.

Pairwise refinements allow information from two different Slices to be combined into a new Slice. Daisy-chaining of pairwise operations using fluent-style interfaces produces a new Slice from multiple Slices. It is often advantageous to combine or prune Slices based on common attributes. For example, two datasets covering a particular region may be merged, or specific event patterns from one Slice may be subtracted from another. We provide this functionality with set operations across features, which include the union, intersection, and difference operators. Two Slice datasets are passed to the set operators as inputs and produce a single Slice as their output. Combined with the spatiotemporal aspects of the data, set operations are a powerful way to transform Slices based on element-wise comparisons. Set operations can be chained, performed for particular features, and can prune or expand the spatiotemporal scope of the resulting Slice.

In Listing 1, we present a short Scala example highlighting the functionality of our Java API. In lines 2 - 5, we initialize a Slice using a Synopsis query for geohash 9jxd and temperature range 230 - 328 (K). Lines 10 - 12 display fluent, chained, instrumentation operations including Slice union and range selection (The remainder of the listing is explained in Section 3.3.1).

Listing 1: Scala code outlining Java API functionality. Slice initialization under geohash and temperature constraints, Slice set and range instrumentation operations, and Slice materialization on Anamnesis with a variety of configuration options.

```
1  // query synopsis for slice
2  var slice1 = Context.synopsisQuery("10.0.0.10",
       4500)
3    .setGeohash("9jxd")
4    .addRange("temperature", 232.0f, 328.0f)
5    .execute();
6
7  // slice2 creation redacted
8
9  // slice instrumentation operators
10 var slice3 = slice1
11   .union(slice2)
12   .selectRange("temperature", 260.0f, 300.0f);
13
14 // slice materialization
15 Context.anamnesisMaterializer("10.0.0.40", 8020)
16   .setShardInterval("temperature", 250, 10)
17   .setFeatureIndexes(Array(2, 1))
18   .setFilename("/user/hamersaw/slice1.csv")
19   .setInflationRate(0.8)
20   .execute(slice3);
```

3.2.1 Maintaining the Lineage of a Slice

Each Slice maintains a lineage construct that encapsulates a full listing of its instrumentation operations. Lineage is organized as a tree where each node represents a single operation. We capture the operation/operand relationship with parent and children nodes, where an operation's operands are the lineage node's children. The leaf nodes in the lineage graph contain the query that resulted in the particular Slice. Each lineage node maintains the entire set of configuration variables for that operation, allowing for unambiguous Slice reconstruction. A sample lineage tree for a Slice is shown in Figure 3. Slice S is constructed by a union of two slices (P and Q) resulting from two separate queries (Query A and B, respectively). Similarly Slice R is constructed by Query C. Slice T is built by the difference between Slice S and Slice R.

The lineage construct serves two purposes. First, a JSON representation is readily available to identify data origins as active Slice counts increase during the iterative analytics process.

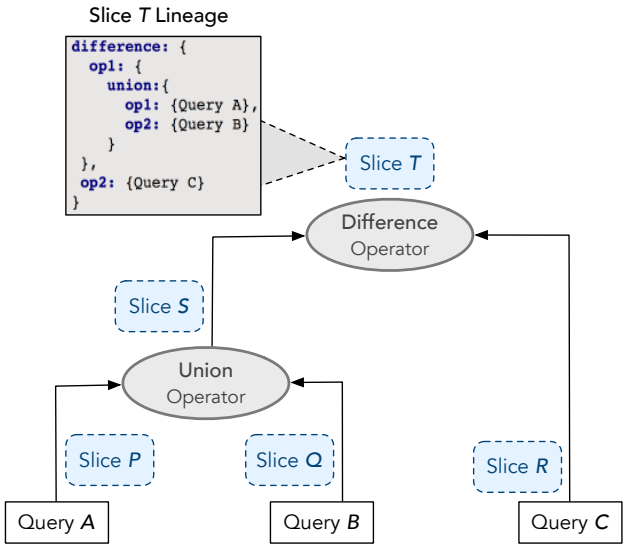


Fig. 3: An example Slice lineage tree and its JSON representation including multiple Synopsis queries and set operations. Lineages provide Slice reproducibility when instrumentation operations are re-executed.

Second, the lineage construct is leveraged to alleviate fault-tolerance overhead. The structure may be reliably stored to disk to persist during system restart or failure. Each lineage describes a unique Slice which is easily reconstructed based on its encapsulated directives in case of data corruption.

3.3 Materializing Slices [RQ-2, RQ-4]

A Slice represents the dataspace of interest and is used as the basis for launching processing tasks. The Slice can be used to generate exploratory datasets that serve as input to analytical engines. Generation of exploratory datasets is governed by a process called *materialization*. A key step in the process is to partition the Slice into shards amenable for dispersion and distributed processing. The materialization process encompasses:

1) Partitioning the Slice into a set of shards.
2) Distributing these shards over the set of available machines.
3) Inflating the shards *in place* based on the specified data generation directives. Our methodology facilitates distributed generation of statistically-representative, synthetic datasets that preserve characteristics of the multidimensional feature space.
4) Encoding records within the inflated shard in myriad formats.

Several customizations are possible for each phase. The same Slice can be materialized differently to produce different exploratory datasets.

3.3.1 Support for Sharding

Depending on the analysis being performed, the set of data items within a dataset may need to be partitioned differently. Consider the following exemplifying scenario: a data scientist is interested in exploring usage patterns and potential cost savings for a smart-grid system. One analysis may target identifying trends in energy consumption to forecast future needs. Here data needs to be partitioned at different temporal scopes such as hours, days, weeks, etc. Another analysis may

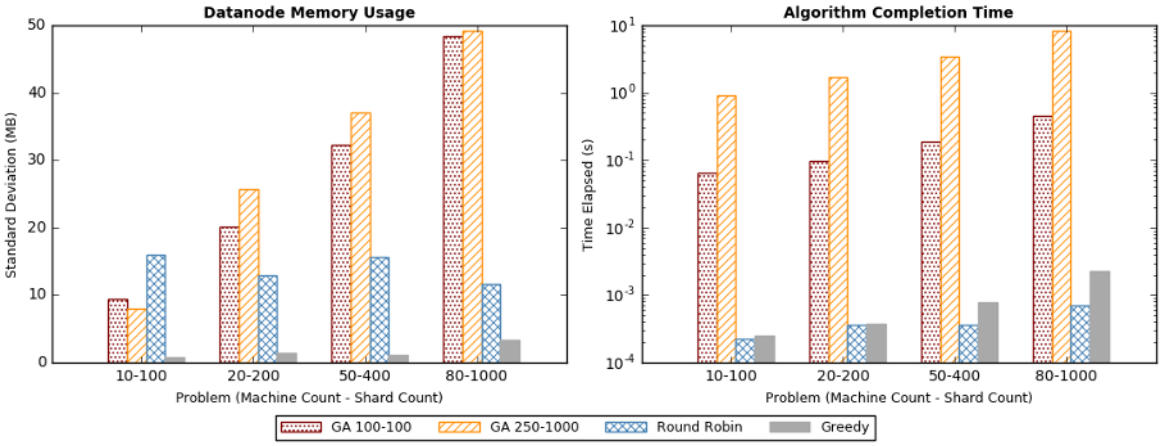


Fig. 4: Shard distribution techniques compared by datanode memory usage standard deviations and algorithm execution time. Techniques including genetic algorithms (with a variety of parameter settings), round-robin, and greedy algorithms. X-axis displayed as "# of machines - # of shards".

seek to understand the impact of weather on energy consumption patterns. This analysis may partition the primary energy consumption dataset based on ranges of energy consumption values and supplement analysis with auxiliary weather data. Finally, another analysis may seek to identify the peaks in energy consumption that coincide with peaks in energy prices, i.e., are energy expenses high because a lot of energy is being consumed or because the consumption is dominated by usage during peak prices?

We support partitioning (or sharding) of slices prior to processing. The sharding process partitions the Slice into a set of non-overlapping tree paths. Each shard comprises one or more tree paths; a given tree path belongs to no more than one shard.

Broadly speaking, a shard can be thought of as grouping tree paths that satisfy a particular broad constraint. A Slice encompasses multiple features of interest and one of these features is designated as the primary key. Once a feature is designated as the primary key, the trees comprising the Slice are reoriented to ensure that the feature chosen as the primary key is now the root node in the forest of trees comprising the Slice.

Across different shards, the primary key takes on different ranges of values. The primary key within a shard satisfies a particular constraint, implicit or explicit. For example, if a user is interested in analyzing monthly rainfall patterns in a given year, the primary key would be the timestamp associated with observations. The tree paths comprising the Slice would be grouped into 12 shards; the first shard would comprise all records for January, the second for February, and so forth. In particular, shards are generated such that they are aligned with the processing being performed so the computations have data locality. Ensuring data locality precludes the need to perform expensive I/O to *pull* data over the network to the computation.

Continuing our example in Listing 1, we materialize the resulting Slice in lines 15 - 20. We define the Slice's shards by a temperature interval with an upper bound of 250 Kelvin and delta 10 (resulting in incremental values at intervals of 10 including 250). We include a subset of the features and re-order them as feature 2, then feature 1. We also set the inflation rate to 0.8 (the resulting synthetic dataset will contain 80% of the total number of observations in the original data).

3.3.2 Distribution of Shards over a Collection of Machines

Exploratory datasets comprise inflated shards that are distributed and pinned in memory. Shards comprising the Slice are distributed and inflated in place. Each shard includes inflation directives specifying the number of observations to be generated for each spatiotemporal scope.

Once a set of shards has been identified, the next challenge is to distribute shards while load balancing the expected processing being performed. The crux of our distribution involves tracking real-time memory utilization statistics across the set of available machines and computing shard sizes to inform distribution of shards.

Using this utilization information, we are able to frame shard distribution as a combinatorial optimization problem where we distribute shards such that nodes have proportionally equal memory utilization; this is equivalent to ensuring that each node holds roughly the same number of records. Given that each record is subject to identical processing, we expect the loads to be balanced. This problem mimics the classic knapsack problem.

Based on efficient multiobjective solutions for the knapsack problem [5] and resource allocation [6] we explored genetic algorithms as a possible solution. The algorithmic parameters (i.e., evolutionary iterations, population size, and mutation probability) allow for fine tuning to optimize different shard distribution scenarios (e.g., many small shards vs. few large shards). In Figure 4 we evaluate performance of genetic algorithms against simple greedy and round-robin shard distribution techniques. Both greedy and round robin algorithms iterate through a list of shards sorted in descending order based on the observation counts in each shard. The greedy algorithm assigns each shard to the machine with the lowest memory utilization at each iteration. The two metrics we use to assess the algorithms are the standard deviation in memory usage across cluster machines and algorithm completion time. A lower standard deviation in memory usage is indicative of evenly distributed shards. Figure 4 shows these two metrics performed on varying problems where the x-axes are represented as "datanode count - shard count" (shard size range is 10MB - 60MB). It shows that both greedy and round robin algorithms continually result in more uniform shard distribution and faster completion times. However, more complex situations may benefit from the genetic algorithm approach.

TABLE 1: Accuracy comparison of synthetically generated observations with the original dataset using feature means, standard deviations, and the Kruskal-Wallis test. Benchmarks on 3 unique datasets show our technique produces statistically representative synthetically generated observations.

Dataset *(Observations)*	Feature *(Unit)*	Mean		Standard Deviation		Kruskal-Wallis P-Value
		Original	Synthetic	Original	Synthetic	
NOAA *(720)*	Temperature *(Kelvin)*	288.1550	288.1838	6.5140	6.5338	0.961
	Humidity *(Percent)*	77.2531	77.1528	19.4883	19.5615	0.9577
NOAA *(85920)*	Wind Gust *(Meters / Sec)*	4.0547	4.0548	2.6368	2.6372	0.9493
	Pressure Surface *(Pascal)*	82280.9230	82279.0882	5558.6149	5562.4838	0.9841
NOAA *(515640)*	Pressure Tropopause *(Pascal)*	20189.2068	20189.7558	1146.2397	1146.9376	0.9223
	Precipitable Water *(Millimeters)*	16.9493	16.9489	3.4992	3.5012	0.9857

3.3.3 Distributed Generation of Exploratory Datasets

Once shards comprising a Slice have been dispersed over a collection of machines, the next step is the distributed generation of exploratory datasets. This involves inflating each shard in place to produce a statistically representative set of records for the portion of the data space represented by the shard. Exploratory datasets serve as input to computations. Two key characteristics underpin the efficiency of processing: distribution and data locality. To ensure data locality during processing, each inflated shard is fully resident on one machine. Furthermore, these shards are inflated in memory to alleviate I/O requirements. Processing shards thus involves no network I/O (because of data locality) and no disk I/O (because of memory-residency).

Each tree path within the shard represents a spatiotemporal scope and maintains: (1) summary statistics for each feature along the path, (2) cross-feature covariances, and (3) the number of observations reported for that spatiotemporal scope. We leverage this information to generate synthetic datasets that are statistically representative of the observed distribution in values for a feature and the observed feature covariances at the particular spatiotemporal scope. During the generation of synthetic datasets, a user may optionally specify the total number of observations. This information is used to proportionally generate observations for each tree path comprising the Slice. Our synthetic data generation process models the observations and information therein as a discrete Gaussian mixture. The discrete Gaussian mixture flexibly approximates a very broad class of multivariate distributions. It describes global-scale variation via variation of mean vectors and covariance matrices across leaves, and local-scale variation via covariance matrices within leaves. An inflated shard comprises a collection of records. Each record represents a multidimensional observation with each dimension corresponding to a feature of interest.

Micro-benchmark: In Table 1, we report on the statistical representativeness of our synthetically generated observations. We profiled 3 unique spatiotemporal dataspaces to assess representativeness. NOAA (720), NOAA (515640), and NOAA (85920) are subsets of the NOAA dataset with durations one week, one week, and one day; geohashes 9xjd, 9xj, and 9x; and containing 720, 515640, and 85920 observations respectively. Observation timestamps may be included in sketch definitions. Therefore, analysis of intervals exceeding one week is unnecessary as data spanning multiple weeks will be placed into separate tree paths of the sketch, and each tree path will generate statistically representative datasets.

The results in Table 1 include two features for each dataset. We note the statistical representativeness was similar for all other tested features. We have provided two separate metrics: (1) combination mean and standard deviation reported for both the original and synthetic datasets; (2) the p-value for the Kruskal-Wallis test. The Kruskal-Wallis test is a statistical technique to determine if two datasets are sampled from different populations. We use standard confidence level 0.05 to refute the null hypothesis that the samples are drawn from different populations. Table 1 shows that all features (spanning all datasets) have similar mean and standard deviation values between the original and synthetically generated observations. Additionally the Kruskal-Wallis test p-value is far larger than the 0.05 value required to refute the null hypothesis. The exploratory datasets are statistically representative of the observed spatiotemporal phenomena.

3.3.4 Encoding Records in Myriad Formats

Since the Slice is data format agnostic, the exploratory dataset may be encoded in myriad formats specified by the user.

Integration with both popular analytics engines and in-house analytics solutions relies on diverse format support. In addition to a number of common format implementations (CSV, binary, etc.,) we have leveraged 3rd party libraries including the NetCDF Java library for Unidata's [7] Common Data Model (CDM) to support a large variety of data formats. A novel advantage our approach has over HDFS is the ability to support binary formats. We are able to guarantee that no individual observation spans multiple blocks. Table 2 displays examples of the diverse set of encoding formats that we support.

3.4 Interoperation with Analytics Engines [RQ-1, RQ-3]

HDFS (Hadoop Distributed File System [2]) is a distributed file system designed for efficient, fault tolerant data storage

TABLE 2: Table displaying supported dataset materialization formats. We leverage 3rd party libraries to support myriad formats. Additionally, our technique enables an HDFS compliant interface for binary formats ensuring blocks are split along record boundaries.

Format	Description
Binary	Sequence of binary floats (one per feature)
CSV	Comma-Separated Values
Protobuf	Google's language agnostic binary data format
Sequence	Mahout / Hadoop data format
GRIB-2	WMO GRIB Edition 2
HDF4	Hierarchical Data Format, version 4
HDF5	Hierarchical Data Format, version 5
netCDF	netCDF classic format
netCDF-4	NetCDF-4 format on HDF-5
NEXRAD-3	NEXRAD Level-III Products
OPeNDAP	Open-source Network Data Access Protocol
SIGMET	SIGMET-IRIS weather radar

and retrieval. HDFS targets the dominant use cases of appends rather than random-writes or in-place updates. Many distributed analytics engines incorporate support for HDFS, including Apache's Spark, Hadoop, Mahout, Flink, and Google's TensorFlow.

In HDFS, files are partitioned into multiple blocks, usually 64MB or 128MB in size. Each block is replicated (default replication factor is 3) among many machines for redundancy and high performance as block reads may occur on any replica. This facilitates distributed analytics allowing a cluster of machines to operate on different portions of a single file in parallel.

An HDFS deployment consists of instances of two applications, the namenodes and datanodes. Namenodes are responsible for all control plane traffic including datanode and block management. The namenode monitors datanode activity and ensures consistent, balanced data replication. Datanodes are provisioned for block storage and efficient retrieval. They receive commands from the namenode to perform block transactions (e.g., initialization, deletion, replication). Data storage and retrieval in HDFS begins with a client contacting a namenode requesting datanode locations. The client then directly connects to each datanode initializing the data transfer.

Canonical HDFS is tightly coupled with on-disk data residency. To read data from disk, it must first be written to it — expensive disk I/O occurs twice. *How can we facilitate data loading into analytical engines without performing disk I/O?* HDFS also includes functionality such as data replication, cache management, corruption detection, etc., that are unnecessary for our problem. Materialization of Slices are transient, with residency tied to the duration of the analytics process. Further, our materialization directives allow users to specify the number of machines to be involved in the analytics process.

We provide seamless integration with analytics engines by leveraging Anamnesis to present an HDFS-compliant interface. Our communication protocol implementations are identical to canonical HDFS serving as a "drop-in" replacement. We have extensively tested Anamnesis interoperability with the native HDFS client v2.8.2, Apache Spark v2.2.0, Hadoop v2.8.2, Mahout v0.13, and TensorFlow v1.7.

3.4.1 Comparison of Anamnesis with Canonical HDFS

We compared Anamnesis with canonical HDFS using RAMDisk storage. RAMDisk provisions a subset of available RAM as an OS managed in-memory hard drive. The two solutions present an identical interface and in-memory analytics opportunities but vary greatly in internal functionality. When integrated into our system we find Anamnesis introduces three main advantages over canonical HDFS with RAMDisk.

- *Native support for Slices:* We leverage the low-level HDFS data transfer API to support data transfer of Slices by breaking up a Slice into multiple HDFS supported blocks. In doing so, the data upload workflow is identical to canonical HDFS except instead of transferring full-resolution data we send Slice shards. Blocks are then stored internally as a set of sketch tree paths, and full-resolution datasets are generated dynamically. These techniques will be explored extensively in further sections.
- *Alleviating system overhead:* Overhead manifests in a number of locations including data replication and data caching. Our lineage construct and Slice materialization schemes allow us to sustain failures with targeted recovery — this obviates the need for replication (though our implementation supports it). Block-level data corruption is remedied by rematerializing affected data, whereas corruptions of sketchlets require Slice redistribution from the Synopsis system. Both techniques result in negligible execution times compared to overall analytics times. Data caching is unnecessary with in-memory analytics systems. Canonical HDFS relies on data caching to provide high performance operations by negating disk I/O for popular portions of the dataspace. Tachyon [8] notes that as a result of Java I/O streams, RAMDisk HDFS fails to bypass caching and requires a separate data copy to memory (even though data is already memory resident in a RAMDisk). This reduces read speeds to almost half of the achievable in-memory speed.
- *Network I/O reduction:* We reduce network I/O in a number of places. First, our native support for sketches vastly reduces bandwidth induced during file creation. Canonical HDFS requires transfer of full-resolution data while Anamnesis is able to transfer compact Slices. Second, we construct and distribute shards based on the proposed analytics. Analytics engines attempt to schedule tasks on machines while ensuring data locality to reduce network I/O. We are able to shard our dataset and tailor data placement to minimize necessary data transfer during analytics. For example, consider the case where we want to generate a histogram for temperatures in intervals of 10. While canonical HDFS needs to aggregate each interval from each datanode, we can ensure the shard for each interval is stored on a single datanode.

TABLE 3: A few HDFS control plane message types that were implemented in Anamnesis to satisfy HDFS compliance.

Operation	Description
addBlock	Add a block to an existing file.
create	Create file.
delete	Delete file.
getBlockLocations	Return locations of datanodes owning block.
mkdirs	Make directories.
rename	Rename a file or directory.
setPermission	Set the permissions on a file or directory.
registerDatanode	Initial datanode registration.
sendHeartbeat	Heartbeat message containing datanode status.
blockReport	Block information (corruption, ownership, etc).

In HDFS, the interprocess RPC-based communication is based on Protobuf [9], Google's protocol buffer implementation. Protobuf uses its own language to define custom message structures. Compilers are implemented for most popular languages to produce native code to serialize/deserialize and interact with messages. As a result Protobuf provides language-agnostic message serialization.

Anamnesis emulates the HDFS architecture with separate namenode and datanode applications. Control plane communication is isolated in the namenode and leverages the aforementioned custom RPC implementation. We implemented 16 different operations for client to namenode communication and 3 operations for datanode to namenode communication. A few example operations are provided in Table 3. The data plane handles transfer of blocks and is present only at the datanodes. While Protobuf facilitate simplicity for RPC, the inherent overhead (Java reflection, additional serialization/deserialization costs) is not suitable for transfer of raw data blocks. Therefore, the HDFS project utilizes a separate binary protocol to read / write blocks at the datanodes. We reimplemented and integrated this protocol into our datanode implementation. Additionally we extended it to include native sketch transfers. This allows us to support canonical HDFS operations along with sketch based read/write operations.

3.4.2 Handling Memory Contention

Our application relies on in-memory working datasets and is therefore subject to memory contention constraints. The challenge of successfully executing on commodity hardware compounds the issue. To combat memory contention we employ two techniques: *just-in-time inflation* and *memory eviction*.

Just-In-Time Inflation: Generation of full-resolution datasets is delayed until requested. Anamnesis stores blocks as a set of sketch tree paths, which are significantly smaller than full-resolution data. This serves to reduce both network I/O and datanode memory usage. All blocks in the system fall under one of two states: *uninflated* or *inflated*. Uninflated blocks contain only the shards that encapsulate the tree paths, whereas inflated blocks additionally contain a full-resolution synthetically generated dataset. Upon data request, an uninflated block transitions to an inflated state by adaptively generating synthetic data for the shard. Our just-in-time inflation ensures that no data is memory resident until it is needed.

Memory Eviction: We periodically monitor datanode memory utilization and trigger targeted memory eviction based on configurable thresholds. For example, if datanode memory is saturated beyond 80% we may attempt to reduce it to 40%. This process iterates over inflated blocks and flags "cold" blocks based on block usage patterns. The aim is to allow popular blocks to remain in memory and be readily available. Blocks flagged for eviction are simply transitioned from the inflated state to uninflated by erasing the full-resolution dataset. Since a block comprises self-describing shards a new, full-resolution synthetic dataset may be generated dynamically for subsequent requests with negligible time and resource costs (as explored in Figure 5).

Figure 6a depicts the Anamnesis data block paradigm of a single datanode. We see that each datanode contains many blocks that fall into one of the two states, inflated or uninflated. In Figure 6b, inflated blocks contain full-resolution data (shown by the array of data in the data portion), whereas uninflated blocks contain an empty data portion. All blocks contain an identifying number (ID) and the set of tree paths comprising the shards that define it. Uninflated blocks become inflated as the data is requested and inflated blocks revert to uninflated as the memory eviction process flags them for removal.

Figure 7 contrasts Anamnesis (with and without just-in-time inflation and memory eviction) and RAMDisk HDFS. All systems were evaluated on a sequence of 50 write and 80 read requests issued over 1000 seconds. Each operation is performed on a subset (approximately 287 MB chunks) of NOAA data. We focus on a single datanode in this experiment. In all plots the x-axis presents the elapsed time of the write/read sequence in seconds and the y-axis is memory usage in MB. The top graph shows Anamnesis (with the aforementioned memory contention handling techniques). In this experiment we cap JVM memory allocation at 10GB and notice the system reaches that at around 500 seconds. The entire sequence successfully completes while Anamnesis' memory usage remains below the memory cap. The constant fluctuations in memory usage are due to periodic JVM garbage collection. The middle graph contains default Anamnesis without any memory contention handling techniques. In this experiment we allocate a maximum of 30GB of memory to the JVM. As memory usage surpasses the system's available RAM, the OS automatically uses the Linux swap disk. We find the constant use of Linux swap disk results in extremely poor performance, and the sequence requires over 5000 seconds to complete. The final graph focuses on RAMDisk HDFS. We allocated a 6GB RAMDisk, which is a maximum allotment on a machine with 12GB of RAM. Since this setup stores data in a RAMDisk, we plot that usage in addition to JVM and Linux swap disk. The system successfully copes with the write/read operation sequence until the 495 second timestamp where the RAMDisk becomes full. All subsequent write operations fail. This experiment shows that existing solutions are inadequate in handling memory contention when coupled with our system. Similarly under real-world use, a memory-resident sketch aligned file system requires memory contention handling to remain useful. We show that the combination of just-in-time inflation and data eviction can successfully mitigate continuous memory contention the system encounters.

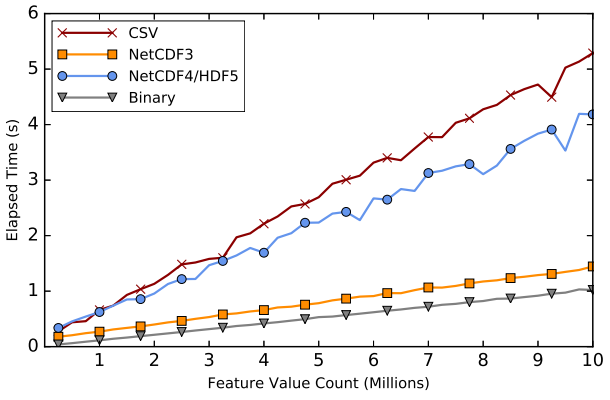


Fig. 5: Linearly increasing synthetic observation generation times based on the number of feature-values requested when performed on a single machine. Binary formats, including NetCDF, outperform human readable formats such as CSV.

4 System Benchmarks

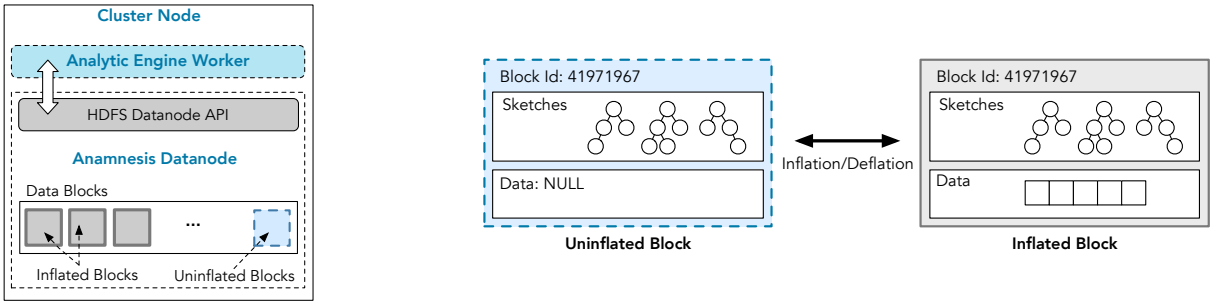
Here we report on our systems benchmarks that profile the suitability of our methodology at scale. With these experiments we aim to highlight interoperability of Anamnesis with popular analytics engines, performance improvements across analytics engines, and accuracy of analytics performed on synthetic datasets. In particular, we:

1) Profile the performance improvements inherit in our system for dataset filtering operations. Worst-case filtering is compared with canonical HDFS.
2) Assess the impact on analyses completion times, network I/O, and data I/O for several commonly leveraged analytic operations. We contrast these attributes for the same analysis code expressed using different analytic engines with and without our optimizations.
3) Contrast the suitability and accuracy of using exploratory datasets when using them for fitting analytical models to the data.

4.1 Experimental Setup

We executed all benchmarks on a set of 50 Hewlett-Packard DL160-G6 machines equipped with a 6-core 2.4 GHz Intel Xeon CPU E5-2620 v3 processor and 16GB RAM. Each machine currently runs Fedora 26 with the 4.15 kernel. For testing we used Hadoop v2.8.2, Spark v2.2.0, and TensorFlow v1.7.

Our benchmarks are performed on two years of data imputed from the NOAA dataset. We have filtered the dataset so each observation contains 56 features uniformly reported in the dataspace. The total size of the dataset is 18TB.



(a) Co-located worker processes of analytical engines request blocks through the Anamnesis HDFS API. A block could either be inflated or uninflated depending on the memory contention level of the datanode.

(b) State transitions between uninflated and inflated blocks based on data requests and memory eviction respectively. These techniques aim to combat memory contention issues of Anamnesis deployments on commodity hardware.

Fig. 6: Data architecture focusing on block storage.

The HDFS setup provisions a single machine as the namenode and the other 49 as datanodes. Storage is restricted to a single 4TB HDD on each machine. Similarly, Anamnesis is deployed with 1 machine as the namenode and the other 49 as datanodes. Both the Anamnesis namenode and datanodes allocate 8GB for the Java heap.

All analytics engines are deployed with a single master located on the same machine as the HDFS/Anamnesis namenode and workers on the other 49 machines. To provide a uniform comparison we configured all analytics engines to leverage the YARN resource manager.

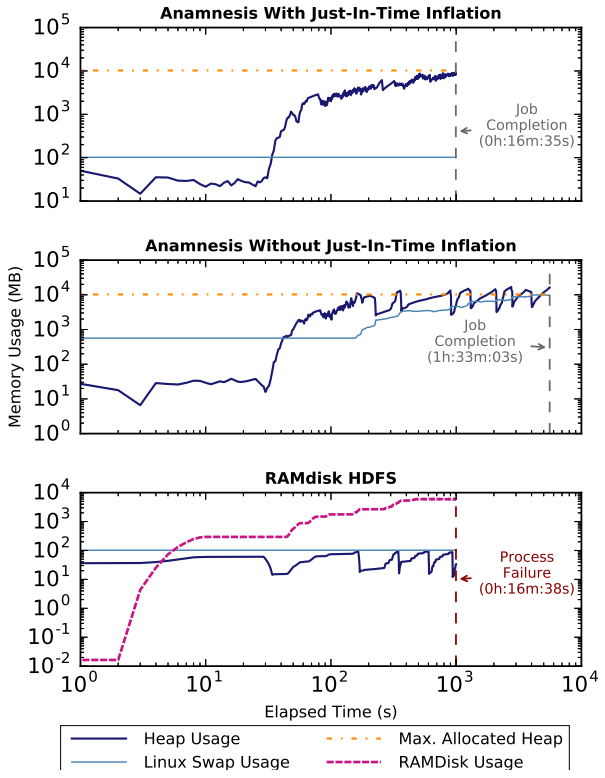


Fig. 7: Comparison of memory contention techniques in Anamnesis with default Anamnesis and RAMDisk HDFS. Each architecture was evaluated under an identical random sequence of read and write operations. We outline Anamnesis' unique ability to successfully complete under memory constraints.

4.2 Dataset Filtering

To support exploratory analytics it is paramount the system can efficiently identify a subset of the dataspace. In the best case, HDFS can tailor efficient data retrieval based on a number of features by manipulating the file hierarchy. For example, placing individual files under directories based on temporal and/or spatial intervals. While this paradigm may simplify filtering on the specified features, inclusion of every feature is impossible. To support high dimensionality dataset filtering within HDFS, iteration over every observation in the dataspace is required. Alternatively, by constructing Sketches using a forest of trees, Synopsis intrinsically presents an index over the data. Therefore, any subset of the dataspace can be identified by fast, in-memory tree traversal.

We present our dataset filtering analyses in Figure 8. In this experiment we compare worst case dataset filtering operations between HDFS and Anamnesis using Apache Spark. In each instance filtering criteria contains temporal (1 month) and spatial (geohash = "9") constraints. The resulting dataset is approximately 400GB. We iteratively increase the base dataset size to showcase our systems ability to filter any size of data with a fixed cost, whereas HDFS increases with size.

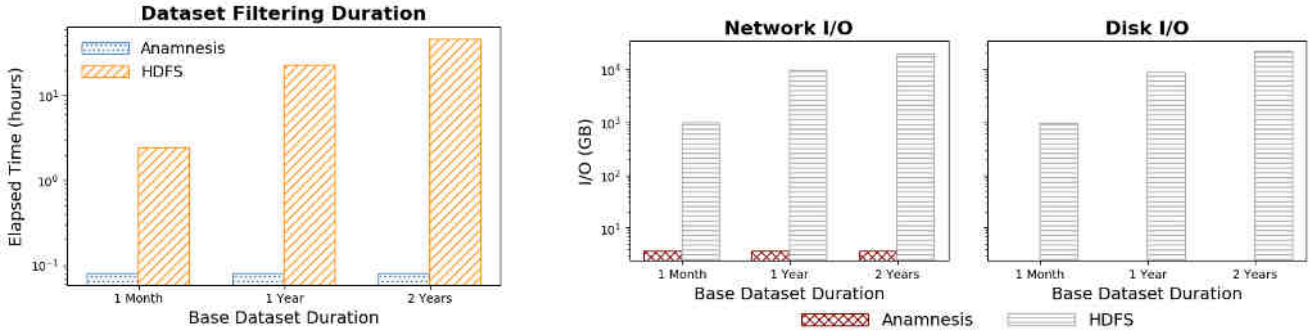
We explore filtering operations duration in Figure 8a. Anamnesis demarcates the times for the Synopsis query and data distribution. We see a 30x, 288x, and 587x reduction in filtering duration when compared to HDFS with base datasets of 1 month, 1 year, and 2 years respectively. Most importantly, the duration to filter using Anamnesis remains constant (approximately 5 minutes) and is independent of the dataset size.

In Figure 8b we provide information on I/O incurred during filtering operations. Network I/O decreases 268x, 2563x, and 5237x when filtering from 1 month, 1 year, and 2 years respectively. This reduction is attributed to (1) all data in transit within our system is being sketched, therefore we benefit from this base data size reduction and (2) HDFS shuffles data between workers during analysis, which is unnecessary with our system.

In all experiments, filtering with HDFS requires the entire dataset to be read from disk. This results in approximately 900GB of disk I/O to filter 1 month and just over 9TB in the case of 1 year. Alternatively, by leveraging the Synopsis sketch Anamnesis performs all filtering actions in memory, resulting in no disk I/O in either case.

4.3 Histogram Generation

A histogram aims to accurately approximate the probability distribution of a single variable. The algorithm requires users



(a) Duration of filtering operations. Anamnesis provides up to 587x reduction by filtering in a fixed time (5 minutes) despite increasing dataset sizes.

(b) Network and disk I/O incurring during filtering operations. Anamnesis shows up to a 5237x reduction in network I/O and disk I/O is completely eliminated.

Fig. 8: Reports on dataset filtering operations comparing Anamnesis with HDFS using Apache Spark over a variety of pre-filtered dataset sizes. Cumulative times reported for Anamnesis demarcate query and data distribution times. All graphs have a logarithmically scaled y-axis, required to visualize the large differences.

to split the range of values into a series of non-overlapping intervals, or "bins". A histogram is produced by plotting bars for each increasing interval on the x-axis, and the number of observations that belong to that bin on the y-axis. This results in an estimation of the kernel density, where smaller bin interval sizes result in a more accurate representation. We have chosen this algorithm as a common exploratory analytics example because it provides a fair, single data pass comparison between Hadoop and Spark.

For this experiment we chose bin interval sizes of length 10, resulting in 14 unique bins with temperature values ranging from 230 to 370 Kelvin. We performed analyses on the 400GB filtered dataset we produced in Section 4.2. The total number of observations in both the HDFS and Anamnesis experiments was 424 million. The combined difference in observation counts across all bins was just 2.8 million or 0.67% of the total number of observations. In Table 4 we present duration, network I/O, and disk I/O of histogram generation contrasting HDFS with Anamnesis using both Spark and Hadoop.

We see analysis duration is reduced by 15x and 34x for Spark and Hadoop respectively. For Anamnesis, the reported duration includes the time required to generate synthetic datasets from the sketched data (ie. transitioning data blocks from the uninflated to inflated states). This time is included but not separately reported because it is insignificant (in the order of milliseconds).

Network I/O is reduced by 20x for Spark and 47x for Hadoop. This is a construct of our data sharding paradigm. When distributing data over the cluster, we tailor data placement for data locality during analytics. In this case, we distribute data based on temperature and ensure each machine only contains data within a distinct interval.

HDFS incurs approximately 350GB and 2.5TB of disk I/O for analysis using Spark and Hadoop respectively. During Spark analytics the entire dataset needs to read into memory. Hadoop had multiple Map/Reduce iterations that each required read and write operations. Alternatively, Anamnesis completely eliminates all disk I/O, as data is memory resident.

TABLE 4: Duration, network I/O, and disk I/O incurred by when generating a histogram using both Spark and Hadoop over HDFS and Anamnesis clusters. Duration is reduced by 15x and 34x for Spark and Hadoop respectively. Additionally, network I/O is reduced by up to 47x and disk I/O is completely eliminated.

Analytics Engine	Storage	Duration	I/O	
			Network	Disk
Apache Spark	Anamnesis	3:26.9	18GB	0GB
	HDFS	52:54.1	362GB	343GB
Hadoop	Anamnesis	2:27.2	27.9GB	0GB
	HDFS	1:24:08.9	1.3TB	2.54TB

4.4 Model Fitting

In Table 5, we use Google's TensorFlow framework [10] to evaluate machine learning models. TensorFlow simplifies use of machine learning for large-scale problems. TensorFlow can be deployed with an HDFS backend. In this experiment, we contrast the suitability of using exploratory datasets versus using full-resolution, on-disk data for model fitting and demonstrate application compatibility. Analysis in Table 1 focused on data accuracy at the feature granularity but failed to validate that the synthetic and original data maintain accurate interfeature relationships.

We trained and evaluated multiple regression models using linear techniques, deep neural networks, and a combination of the two. We used two datasets that both contained 6 features. The first was 1 day with geohash "du" and the second 1 week with geohash "9x". The datasets were partitioned into training (80%) and evaluation (20%). The synthetically trained models were trained on the 80% synthetically generated data but evaluated on the same 20% original dataset that the original trained models were evaluated on.

Our models were built to predict a single target based on the other 5 features. The average loss reported value is the Root Mean Squared Error function commonly used in training regression models. The value is computed by averaging the squared difference between predicted and actual values and calculating the square root value. We report on 6 experiments, noting that other experiments showed similar results and similar accuracy between a variance of variables, model types, and datasets. The results show no significant difference between the synthetic and original datasets.

5 Related Work

Many popular relational and NoSQL database systems have extensions supporting spatial queries. [11] and [12] provide

TABLE 5: Root mean square error comparisons on regressors developed using Google's TensorFlow framework between original data (RAMDisk HDFS) and synthetically generated observations (Anamnesis). Experiments were performed on 2 datasets with varying spatiotemporal scopes and incorporating 6 features, where 5 were chosen to predict the 6th.

Dataset *(Observations)*	Feature *(Unit)*	TensorFlow Estimator	Root Mean Squared Error	
			Original	Synthetic
NOAA *(85920)*	Pressure Surface *(Pascal)*	Linear Regressor	5341.73	5386.72
	Precipitable Water *(Inches)*	Linear Regressor	4.21	5.33
NOAA *(5255840)*	Temperature *(Kelvin)*	DNN/Linear Combined	15.36	16.22
	Humidity *(Percent)*	DNN/Linear Combined	21.27	21.17
	Pressure Tropopause *(Pascal)*	Deep Neural Network Regressor	2404.16	2159.68
	Wind Gust *(Meters / Sec)*	DNN/Linear Combined	3.41	3.97

spatial query support for points, lines and polygons for mariadb and postgres respectively. Geomesa [13] combines three elements of geometry and time using a custom geohash implementation to provide queries using Google Cloud Bigtable [14], Apache HBase [15], Cassandra [16], and more. Some efforts [17], [18] provide spatial support for the MapReduce framework. Unlike these systems, our implementation natively supports efficient spatiotemporal queries provided by the Synopsis sketch. This alleviates overheads in the extension subsystem.

Distributed systems are trending towards in-memory data storage, mitigating disk I/O in favor of magnitudes faster RAM. Redis [19] provides extensive functionality for a key-value storage framework but large datasets are unlikely to fit in a clusters RAM. Apache's Ignite [20] relies on a storage hierarchy to facilitate disk persistence of in-memory datasets enabling larger data sizes and failure resiliency. In-memory systems suffer from the relatively low RAM availability of commodity hardware. This greatly reduces deployability in a variety of environments. While some systems are adopting a storage hierarchy we use just-in-time inflation and memory eviction techniques to ensure full-resolution data is only in memory as needed. This allows us to provide an entirely in-memory storage system without incurring expensive disk I/O.

Analytics over voluminous data have been driven by distributed engines that interface with HDFS. Hadoop [21] provides an open source Map-Reduce framework. Mahout [22] extends the MapReduce framework to include linear-algebra specific functionality. Spark [23] and SparkSQL [24] offer various analytical operations on in-memory RDDs (Resilient Distributed Datasets). Additional tools [25] enable machine learning over HDFS. Anamnesis implements an identical communication protocol to HDFS. We support integration with all of the aforementioned tools providing an in-memory storage system able to reliably alleviate memory contention for iterative analytics.

Extensions on distributed analytics engines enable spatiotemporal queries. The SpatialHadoop framework [26] supports indexing spatial data using grid files, R-trees and R+-trees to perform efficient data retrieval. Hadoop-GIS [27] extends this functionality to provide relation queries such as contains, intersects, distance, etc. Geospark [28] maintains R-trees and Quadtrees to create indexes on 4 types of RDDs: PointRDD, RectangleRDD, CircleRDD, and PolygonRDD. SpatialSpark [29] introduces a distributed spatial join over datasets. STARK [30] builds on both Geospark and SpatialSpark to include persistent indexes and additional analytics operations including nearest neighbors and clustering. These frameworks rely on indexing spatiotemporal features during data writes to efficiently evaluate queries. SpatialHadoop and Hadoop-GIS target the storage layer, whereas Geospark, SpatialSpark, and STARK work during analytics. Alternatively our system provides this functionality at a more conceptual phase. Slices provide the ability to initialize and configure datasets based on a variety of spatial and temporal properties. Our system supports identical operability without incurring significant resource utilization. Additionally all of the Spark extensions may easily integrate with our system to provide any missing functionality.

Simba [31] introduces native spatial querying operators into Spark SQL [24] and Dataframes. This extended API allows users to implement geospatial analytic tasks directly on Spark. But our approach offers more benefits especially for streaming datasets. It enables performing ad-hoc analytics on historic data without having to store the entire stream, which is space inefficient.

Tao et al. [32] support distinct counting queries over single attribute spatiotemporal data. Their work extends the previous work on aRB trees [33] to maintain the spatiotemporal index (using R trees and B trees to implement spatial and temporal indexing respectively). It also uses a sketch based on FM algorithm [34] to maintain the distinct objects for individual spatiotemporal scopes. In comparison, Synopsis (our underlying sketching algorithm) supports multifeature spatiotemporal data streams and a wider range of queries. On the other hand, most of the design principles presented in this work to achieve efficient data analytics are applicable to other sketching frameworks.

There have been several efforts to improve HDFS MapReduce performance by allowing programmers tighter control over data caching and placement to facilitate in-memory data local analytics [35], [36], [37]. Users may specify a subset of the dataspace to be read into cache. Additionally, MapReduce operations requiring multiple iterations may keep intermediary datasets cached instead of disk writes. While these systems combat network and disk I/O at the analytics level our approach targets data storage. We eliminate program cache coherency issues by making all working sets memory resident. This ensures mitigation of all disk I/O.

Recent work has extended the HDFS codebase to make in-memory HDFS analytics available. Triple-H [38] integrates a data hierarchy paradigm leveraging diverse storage mediums (RAM, SSD, HDD) that make working datasets memory-resident. This allows the system to cope with much larger datasets than the collective memory capacity of the cluster. Tachyon [8] relies on data lineages and data checkpointing to store working datasets in memory while deleting temporary intermediary datasets. Using their lineage construct, datasets may be re-computed in the background based on access and resiliency requirements. Our work also produces an in-memory, HDFS compliant analytical interface. The advantage of our solution is the massive reduction in network I/O. The system transports data sketches instead of full-resolution data, result-

ing in a 50x reduction in network I/O during analytics.

Generating representative synthetic datasets based on original data has been explored in statistical work. MUNGE [39] and many SMOTE variants [40], [41] under sample the minority class and over-sample the majority dataset to achieve representative synthetic datasets. These techniques focus on capturing and reproducing representative data outliers. This in-depth functionality is unnecessary in our system. The Synopsis sketch successfully captures data outliers by defining various data boundaries in its tree paths. We can generate highly configurable, representative synthetic datasets based on this representation.

6 Conclusion and Future Work

In this work we have presented Anamnesis, our methodology to facilitate fast analytics over spatiotemporal data. To our knowledge, Anamnesis is the first sketch-aligned, HDFS-compatible file system. Below we address our initial research questions.

RQ-1: We significantly alleviate inefficiencies resulting from the speed differential across the memory hierarchy. Anamnesis facilitates memory-residency of exploratory datasets, makes frugal use of memory, and avoids all disk I/O. We employ just-in-time inflation and memory eviction techniques to enable continuous analytics without memory contention.

RQ-2: Effective sharding vastly reduces network I/O during analytics. By tailoring dataset distribution based on the proposed analytics, we allow analytic engines to successfully schedule tasks while preserving data locality. Additionally the process of identifying sketches and generating synthetic datasets enables efficient exploratory analytical operations without incurring unnecessary network I/O during dataset filtering and refinements.

RQ-3: Anamnesis is an in-memory implementation compatible with canonical HDFS. This allows interface with diverse analytical engines. Our implementation facilitates adoption into existing workflows by serving as a "drop-in" replacement for any HDFS deployment.

RQ-4: A combination of the techniques described in this work (in-memory analytics, reduced network I/O, etc.) ensures fast analytics completion times. We have shown Anamnesis' ability to filter datasets with a fixed cost and offer evaluation performance gains of up to 47x when executing a diverse set of analytical operations over HDFS.

6.1 Future Work

Zero copy analytics: Our approach requires in-memory dataset duplication for most analytics. For example, analytics using Apache Spark require the data to be Anamnesis resident (one copy) and read into a Spark RDD (second copy). Using shared memory with a custom Spark connector we should be able to operate on a single in-memory copy. Anamnesis will materialize blocks into a shared memory segment which may then be read by the custom Spark connector.

Resource usage sharding optimization: Our algorithm for distribution of Slice shards only accounts for memory usage at each datanode. Intuitively we believe that analytics performed on similar sized data will take similar durations. While it proven a great starting point, this may not hold in all scenarios. For example, if a particular datanode contains a disproportionate share of popular (or unpopular) data. We propose including more system resources in the data sharding algorithm for better load-balancing. Leveraging CPU utilization, and network I/O statistics may provide additional insight into utilization beyond memory usage.

Data interpolation: The system can only produce synthetic observations based on statistical representations of known datasets. The ability to leverage relationships between known datasets to produce synthetic observations for an unknown dataset is beneficial to myriad analytics. For example, we have a 2018 dataset for Denver, Boulder, and Fort Collins. If we have April data for Fort Collins and Boulder, can we leverage the relationships between between all of the datasets to interpolate a dataset for April in Denver? We aim to adopt proven techniques to accurately produce the proposed datasets with our system.

6.2 Experimental Reproducability

The source code for all applications used in this work are hosted in public git repositories. Anamnesis is located at https://github.com/hamersaw/anamnesis and Synopsis at https://github.com/thilinamb/synopsis. These are Java projects using the Gradle and Maven build systems for Anamnesis and Synopsis respectively.

Both the NOAA weather and EPA air quality datasets are publicly available. The NOAA dataset is available in a number of formats. We specifically used the GRIB format available at https://nomads.ncdc.noaa.gov/data/namanl. The EPA dataset is available at https://aqs.epa.gov/aqsweb/airdata/download_files.html#Raw.

Acknowledgments

This research was supported by the US Dept. of Homeland Security [HSHQDC-13-C-B0018, D15PC00279]; the National Science Foundation [ACI- 1553685, OAC-1931363]; and the Environmental Defense Fund.

References

[1] T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara, "Synopsis: A distributed sketch over voluminous spatiotemporal observational streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2552–2566, 2017.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system: Mass storage systems and technologies (msst) 2010 ieee 26th symposium on," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010.

[3] G. Niemeyer. (2008) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[4] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4(3), pp. 419–420, 1962.

[5] C.-M. Lin and M. Gen, "Multi-criteria human resource allocation for solving multistage combinatorial optimization problems using multiobjective hybrid genetic algorithm," *Expert Systems with Applications*, vol. 34, no. 4, pp. 2480–2490, 2008.

[6] M. J. Alves and M. Almeida, "Motga: A multiobjective tchebycheff based genetic algorithm for the multidimensional knapsack problem," *Computers & operations research*, vol. 34, no. 11, pp. 3458–3470, 2007.

[7] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[8] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–15.

[9] P. Buffers, "Googles data interchange format," 2011. [Online]. Available: https://developers.google.com/protocol-buffers/

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[11] E. Katsikaros, "Towards the universal spatial data model based indexing and its implementation in mysql," *Kongens Lyngby*, 2012.

[12] R. Obe and L. Hsu, "Postgis in action," *Geoinformatics*, vol. 14, no. 8, p. 30, 2011.

[13] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "Geomesa: a distributed architecture for spatio-temporal fusion," in *Geospatial Informatics, Fusion, and Motion Video Analytics V*, vol. 9473. International Society for Optics and Photonics, 2015, p. 94730F.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[15] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

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

[17] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Spatial queries evaluation with mapreduce," in *2009 Eighth International Conference on Grid and Cooperative Computing*. IEEE, 2009, pp. 287–292.

[18] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *International Conference on Scientific and Statistical Database Management*. Springer, 2009, pp. 302–319.

[19] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[20] S. Bhuiyan, M. Zheludkov, and T. Isachenko, "High performance in-memory computing with apache ignite," 2017.

[21] A. S. Foundation, "Apache hadoop," 2018.

[22] S. Owen and S. Owen, "Mahout in action," 2012.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proc. of the International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[25] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[26] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1352–1363.

[27] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[28] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015, p. 70.

[29] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*. IEEE, 2015, pp. 34–41.

[30] S. Hagedorn, P. Gtze, and K.-U. Sattler, "The stark framework for spatio-temporal data analytics on spark," *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[31] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1071–1085.

[32] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias, "Spatio-temporal aggregation using sketches," in *Proc. of the Intl. Conference on Data Engineering*, March 2004, pp. 214–225.

[33] D. Papadias, Y. Tao, P. Kanis, and J. Zhang, "Indexing spatio-temporal data warehouses," in *Proc. of the Intl. Conference on Data Engineering*, 2002, pp. 166–175.

[34] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[35] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[36] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM international symposium on high performance distributed computing*. ACM, 2010, pp. 810–818.

[37] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3r: increased performance for in-memory hadoop jobs," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.

[38] N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. Panda, "Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 101–110.

[39] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 535–541.

[40] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-smote: a new over-sampling method in imbalanced data sets learning," in *International Conference on Intelligent Computing*. Springer, 2005, pp. 878–887.

[41] C. Bunkhumpornpat, K. Sinapiromsaran, and C. Lursinsap, "Safe-level-smote: Safe-level-synthetic minority over-sampling technique for handling the class imbalanced problem," in *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 2009, pp. 475–482.

Daniel Rammer is a Ph.D. candidate in the Department of Computer Science at Colorado State University. His research interests involve big data and distributed analytics. In particular systems approaches for distributed analytics over voluminous spatiotemproal data in myriad domains. Email: rammerd@rams.colostate.edu

Thilina Buddhika is a Ph.D. candidate in the Computer Science department at Colorado State University. His research interests are in the area of real time, high throughput stream processing specifically targeted to environments such as Internet of Things (IoT) and health care applications. Email: thilinab@cs.colostate.edu

Matthew Malensek is an Assistant Professor in the Department of Computer Science at the University of San Francisco. His research involves big data, distributed systems, and cloud computing, including systems approaches for processing and managing data at scale in a variety of domains, including fog computing and Internet of Things (IoT) devices. Email: mmalensek@usfca.edu

Shrideep Pallickara is an Associate Professor in the Department of Computer Science and a Monfort Professor at Colorado State University. His research interests are in the area of large-scale distributed systems. He received his Masters and Ph.D. degrees from Syracus University. He is a recipient of an NSF CAREER award. Email: shrideep@cs.colostate.edu

Sangmi Lee Pallickara is an Associate Professor in the Department of Computer Science at Colorado State University. She received her Masters and Ph.D. degrees in Computer Science from Syracuse University and Florida State University, respectively. Her research interests are in the area of large-scale scientific data management. She is a recipient of the NSF CAREER award. Email: sangmi@cs.colostate.edu