DISSERTATION

HARNESSING SPATIOTEMPORAL DATA CHARACTERISTICS TO FACILITATE LARGE-SCALE ANALYTICS OVER VOLUMINOUS, HIGH-DIMENSIONAL OBSERVATIONAL DATASETS

Submitted by

Daniel P. Rammer

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2021

ABSTRACT

HARNESSING SPATIOTEMPORAL DATA CHARACTERISTICS TO FACILITATE LARGE-SCALE ANALYTICS OVER VOLUMINOUS, HIGH-DIMENSIONAL OBSERVATIONAL DATASETS

Spatiotemporal data volumes have increased exponentially alongside a need to extract knowledge from them. We propose a methodology, encompassing a suite of algorithmic and systems innovations, to accomplish spatiotemporal data analysis at scale. Our methodology partitions and distributes data to reconcile the competing pulls of dispersion and load balancing. The dispersion schemes are informed by designing distributed data structures to organize metadata in support of expressive query evaluations and high-throughput data retrievals. Targeted, sequential disk block accesses and data sketching techniques are leveraged for effective retrievals. We facilitate seamless integration into data processing frameworks and analytical engines by building compliance for the Hadoop Distributed File System. A refinement of our methodology supports memory-residency and dynamic materialization of data (or subsets thereof) as DataFrames, Datasets, and Resilient Distributed Datasets. These refinements are backed by speculative prefetching schemes that manage speed differentials across the data storage hierarchy. We extend the data-centric view of our methodology to orchestration of deep learning workloads while preserving accuracy and ensuring faster completion times. Finally, we assess the suitability of our methodology using diverse high-dimensional datasets, myriad model fitting algorithms (including ensemble methods and deep neural networks), and multiple data processing frameworks such as Hadoop, Spark, TensorFlow, and PyTorch.

ACKNOWLEDGEMENTS

The merits of a Ph.D. are not restricted to innovative research. Rather, growth in character, in capabilities, which drive future progress may be equally important. It is essential to reflect on this growth, and in doing so the understanding that acheivements, which are often illustrated as individual accomplishment, may be equally attributed to the influence and guidance of others. Sir Isaac Netwon acutely summarizes this, "if I have seen further than others, it is by standing on the shoulders of giants".

I owe a great debt of gratitude to my advisor Shrideep Pallickara who, in the throes of my academic fallout, adopted me into passionate mentorship; providing kind instruction and guidance. I hope to move forward exemplifying the same inspirational dedication. Similarly, I would like to recognize my co-advisor, Sangmi Lee Pallickara, whos humble insight has had untold impact on my work.

I am thankful for the support of my parents, Scott and Gwen, who under exaggerated modesty admit they "don't understand a single word" of this dissertation. Their precedent and imparted values ensured the success of my siblings and myself. I am imparcially appreciative of my wife, Kimberly, who for reasons still under investigation has continually offered loving, steadfast support; being the rock to ground me and sharing in every success and failure of this journey.

Finally, I have been blessed with the selfless guidance of passionate educators and coaches who, realizing my potential, shepherded me to opportunities which may not have otherwise been recognized. The positive impact on my development can not be overstated. There are far too many to explicitly identify but I would like to specifically name Terry Gross, Ed Smrecek, John Zupanc, and David Furcy.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
LIST OF TABLES vi
LIST OF FIGURES vii

Chapter 1 Introduction 1
1.1 Research Challenges 2
1.2 Research Questions 3
1.3 Approach Summary 4
1.4 Expected Contributions 6
1.5 Experimental Setup 6
1.5.1 Datasets 7

Chapter 2 Related Work 10
2.1 Lossy Compression and Data Sketching 10
2.2 Spatial Partitioning and Indexing 11
2.3 Distributed Storage 12
2.4 Distributed Analytics 14
2.5 Distributed Model Training 14
2.6 Discussion 16

Chapter 3 Partitioning Spatiotemporal Data 17
3.1 Sketching Datasets 18
3.1.1 Groundwork Phase 19
3.1.2 Supported Multiple Sketches Over the Same Data 19
3.2 Spatiotemporal Block Indexing 21
3.3 Reconciling Differences in Image Attributes 22

Chapter 4 Distributed Storage 26
4.1 Processing Sketched Data 27
4.2 Distributed Hash Tables 28
4.3 Dispersion-Informed Data Replication 30

Chapter 5 Performant Data Retrieval 33
5.1 Enumerating Sketches 34
5.1.1 Materializing Slices 35
5.1.2 Encoding Records in Myriad Formats 36
5.1.3 Instrumentation of Slices 37
5.1.4 Maintaining the Lineage of a Slice 40
5.1.5 Integration With Analytics Engines 41
5.1.6 Alleviating Memory Contention 43

5.2 Filtering Spatiotemporal Data Blocks 48
5.2.1 HDFS Compliance 49
5.3 Spatiotemporal Image Queries 53
5.3.1 Complementing Dataspace Features 54

Chapter 6 Support for Diverse Analytics 56
6.1 Data Wrangling Operations 56
6.2 Scheduling with Data Locality 60
6.3 Incremental Operation Optimizations 62
6.4 Comparison with Canonical Spark and HDFS 64
6.4.1 Histogram Generation 64
6.4.2 Complex Dataset Filtering 65

Chapter 7 Effective Model Training 71
7.1 Training with Sketched Data 71
7.1.1 Data Sampling 73
7.1.2 Model Accuracy 73
7.2 Orchestration of Deep Learning Workloads 76
7.2.1 Duration 78
7.2.2 Resource Utilization 79
7.2.3 Model Accuracy 81
7.2.4 Spatial Transfer Learning 82

Chapter 8 Conclusions 85
8.1 Future Work 87

Bibliography 89

LIST OF TABLES

1.1 Observational datasets evaluated in this work. 8
1.2 Descriptions of satellite imagery datasets used in this work. 9

4.1 Image partition distribution metrics. 29

5.1 Example of control plane messages to satisfy HDFS compliance. 34
5.2 Accuracy of synthetic observations generated from sketched data. 36
5.3 Example of available sketched data materialization formats. 37
5.4 Spatiotemporal filtering operands that may be embedded within Atlas' request URLs. 50
5.5 The bit sequence index layout of Atlas' HDFS block ID extensions. 51
5.6 Comparison of Atlas with canonical HDFS for spatiotemoral dataset filtering. . 52

6.1 Unary operation spatial extensions to Apache Spark. 57
6.2 Binary operation spatial extensions to Apache Spark. 58
6.3 Performance evaluation of unary spatial operations within Spark. 60
6.4 Performance evaluation of binary spatial operations with Spark. 60

7.1 Duration to train models comparing workload orchestration architectures. . . . 78
7.2 Network I/O incurred during model training over spatially partitioned images. 81
7.3 Model accuracy comparing distributed and partitioned orchestration schemes. 81

LIST OF FIGURES

2.1 Partitioning the globe using quadtiles, each additional character produces 4 subregions. 12

3.1 GPU training memory requirements in logarithmic scale, undivided images are 100 times larger than memory capacity. 17
3.2 Dataset compaction using the Fennel sketch. 20
3.3 Spatiotemporal block indexing within Atlas framework. 22
3.4 Atlas block indexing duration. 23
3.5 An example of the image 'fill' procedure. 24
3.6 Spatiotemporal image partitioning duration. 25

4.1 Evaluation of data sketch distribution techniques. 28
4.2 Data block cluster distribution within Atlas. 30
4.3 Number of hosts for each unique geohash in Atlas distribution. 31

5.1 Duration to generation synthetic observations from sketched data in myriad formats. 38
5.2 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. 41
5.3 Dataset filtering metrics comparing Anamnesis with HDFS using Apache Spark. 44
5.4 Memory-based data block storage using sketched data. 45
5.5 Evaluation of memory contention techniques comparing Anamnesis with RAMDisk HDFS. 47
5.6 Atlas' namenode architecture and maintenance of spatiotemporal indices. . . . 48
5.7 The Atlas architecture highlighting the hierarchy of provided protocols and application interfaces. 49
5.8 A sample HDFS block ID broken down according to the Atlas spatial encoding scheme. 51

6.1 AtlasSpark's integration into the analytical hierarchy. 57
6.2 Data partitions for Atlas analytics using our spatial extensions in Spark. 62
6.3 Duration, disk I/O, and network I/O incurred during histogram evaluation over a single dataset feature comparing Atlas with HDFS. 65
6.4 Duration of spatially bounded range queries for AtlasSpark. 66
6.5 Evaluation of cluster memory utilization during range queries in AtlasSpark. . 67
6.6 Disk I/O for spatially bounded queries comparing AtlasSpark, GeoSpark, and Magellan. 68
6.7 Network I/O incurred during spatial range queries using AtlasSpark, Magellan, and GeoSpark. 69

7.1 Data sampling metrics comparing sketched data with on-disk. 72

7.2 Comparisson of data sampling durations for sketched and on-disk data. 74
7.3 Accuracy of models trained on samples from actual and sketched data. 75
7.4 Architecture of distributed workload orchestration techniques. 77
7.5 Representative CPU utilization during model training with various orchestration schemes. 79
7.6 Representative GPU utilization during model training with various orchestration schemes. 80
7.7 Examples of NDMI and SpAGAN predictions visualizing the accuracy of our partition-based training approach. 82
7.8 The effects of transfer learning on training the NDVI model. 83

Chapter 1

Introduction

Spatiotemporal data, encompassing both observational and hyperspectral imagery, has experienced exponential growth in the past decade. This may be attributed to falling costs of sensing environments, improvements in the quality and reliability in system resources, and progress in research efforts for storage and analytics at scale. These data have application in myriad domains including atmospheric sciences, urban planning, epidemiology, and environmental and ecological monitoring.

The data by itself is relatively invaluable; rather, the benefit is the insights they contain. These insights often manifest as relationships within or between data (i.e., inter-feature or inter-domain correlations) and are identified using diverse analytics tools including complex modeling techniques among others. Operating over large data is computationally expensive and the challenges in gleaning these insights intensify relative to the volumes and diversity of data. Therefore, novel solutions are required in the realms of data storage and analytics to facilitate analytical performance and accuracy.

These data progressions have coincided with increased efforts in storage and analytic research. We see distributed file systems which scale to manage petabytes of data and NoSQL databases pushing CAP theorem limits. The popularization of distributed analytics suites enable performant, diverse operations using commodity hardware. Similarly, the emergence of open-sourced machine learning frameworks are fueling an outbreak of deep learning applications.

Many distributed frameworks provide generalized solutions – a jack of all trades, master of none. As such, work on specific domains is often built as layers above existing technologies. We believe these solutions, being subject to constraints of the parent system, overlook myriad improvements affecting performance and accuracy of analytical results. For example, spatiotemporal indexing data structures are often build dynamically during analytics instead of during data ingestion. The intrinsic nature of incremental analytics means this indexing operation has dire implications for large datasets which do not fit in RAM.

This dissertation aims to design, implement, and evaluate solutions for distributed storage and analytics of spatiotemporal data at scale. We facilitate low-level performance improvements by implementing optimizations as first class objectives into distributed frameworks. Our solutions adopt open-sourced, community-driven communication protocols to seamlessly integrate into existing workflows. We architect distributed file systems which use lossy compression algorithms for more efficient utilization of the memory hierarchy, enabling in-memory analytic speeds over large datasets. Our data distribution techniques ensure scheduling of diverse analytics tasks with data locality, mitigating network I/O. We have integrated common spatiotemporal analytic operations (i.e., intersects, within, etc) into the distributed analytics ecosystem. Finally, our deep learning efforts leverage spatiotemporal data partitioning properties to more effectively orchestrate deep learning workloads. It is important to note that while these solutions are implemented under the guise of spatiotemporal data they provide insight into optimizations for diverse domains.

1.1 Research Challenges

The evolution of spatiotemporal data offers unprecedented opportunity for valuable insight. However, processing these data introduces several key challenges:

- **Data Volumes and Diversity:** The proliferation of sensing instrumentation and relatively high data resolutions and dimensionality contribute to petabyte-scale datasets. These quickly outpace storage and processing capacity of a single machine. The increasing diversity in data formats compounds challenges in coping with data at scale.

- **Performance Difference in I/O Subsystem:** Disk access performance is several orders of magnitude slower than memory access. The analytics process is dominated by costs associated by I/O. Therefore, datasets, or portions thereof, must be memory resident for performant analytics. However, dataset volumes surpass RAM in commodity clusters.

- **Data Movements:** Staging data in a distributed cluster requires a decentralized, deterministic paradigm. However, it is impossible to predict data access patterns from future analytics. Therefore, analytics often trigger data movement between cluster hosts significantly degrading performance and introducing unnecessary system bottlenecks.

- **Interoperation with Distributed Frameworks:** The speed with which distributed frameworks are introduced means interoperation between suites is difficult. Solutions implemented without these constraints in mind risk siloing of their progress. Therefore, integration into existing workflows requires implementation in adherence to well-defined community protocols.

1.2 Research Questions

The goals accompanying this work have driven us to formulate the following research questions:

1. How can we build systems which scale to handle growing dataset sizes and reporting formats? *Solutions must ensure load-balanced data distribution, support for efficient analytic scheduling, and targeted, sequential disk access over diverse data.*

2. How can we improve performance, while maintaining accuracy, of analytics through more effective use of CPU / GPU, memory, and disk and network bandwidth? *Evaluation metrics include execution durations, result accuracy, system resource utilization (i.e., memory, CPU, GPU), and bandwidth utilization (i.e., disk and network I/O).*

3. How can we develop solutions which integrate into existing workflows? *Implementations without interoperation risk siloing novel solutions. The promote widespread application solutions must seamlessly interoperate with existing tooling.*

1.3 Approach Summary

Our approach for harnessing spatiotemporal data characteristics to improve storage and analytics performance at scale includes improvements at all levels of the processing stack. Specifically, we (1) partition input data on spatiotemporal attributes, (2) effectively distribute data for spatiotemporal processing, (3) provide performant, interoperable data retrieval, (4) support diverse spatiotemporal analytic operations, and (5) design and implement solutions for more efficient model training over spatiotemporal workloads.

The efficiency of processing spatiotemporal data can by directly correlated with the ease of data access. However, we see that reporting schemes seldom align with spatiotemporal data access patterns. Therefore, staging data includes a pre-processing phase which partitions data while constructing concise indices. We support a collection of data staging paradigms to cope with the diversity in data types. Data files may be partitioned into data blocks with contiguous spatiotemporal extents to facilitate targeted, sequential disk access. Observational data is well suited for data sketching algorithms, a brand of lossy data compression, facilitating dataset size reductions measurable in orders of magnitude which driving in-memory analytics speed. Hyperspectral imagery is partitioned along spatiotemporal bounds with separate processes to reconcile occlusions based on spatiotemporal gaps or cloud coverage.

The increase in data volumes quickly outpace storage and / or processing capacity of a single machine. Distributing the dataspace across a collection of machines is a common solution. However, it introduces intrinsic challenges in load-balancing subsequent analytics commensurate of system resources. We employ distributed hash tables (DHTs), which are a proven technique for dynamic scaling. Our solutions are keyed on spatiotemporal properties for load-balanced distribution with collocation of spatiotemporal extents, mitigating expensive data movements during analytics. Additionally, our proposed dispersion-informed data replication scheme extends current replication schemes to ensure simultaneous distribution and collocation of spatiotemporal extents. This allows diverse analytics to be orchestrated with data locality.

The speed of innovation in this space is both a blessing and a curse. Frameworks introduced with orthogonal protocols, which are inoperable with existing solutions, risk siloing progress. We implement solutions which adhere to open-sourced, community-driven protocols for seamless integration into existing workflows. A main component in our approach is incorporating robust URL-based spatiotemporal queries into file-based storage solutions. Similarly, we support performant data ingestion, generation, and processing capabilities for a diverse collection of data formats.

Supporting diverse operations over large data is extremely difficult. As such, optimizations for domain-specific analytics are often left unexplored. We propose extensions to existing distributed analytic frameworks to provide accurate, performant spatiotemporal support. These extensions are implemented as low-level optimizations, providing improvements in data distribution and analytics scheduling for spatiotemporal workloads. They include support for geometric data types (e.x., points, lines, and polygons) in addition to a rich suite of geometric operations (e.x., contains, intersects, distance, etc).

Machine learning offers new and interesting opportunities for dataspace exploration, where predictive analytics introduces new insights. We demonstrate techniques for effective machine learning over spatiotemporal data. These include sampling approaches over sketched data, offering accurate, in-memory performance for over-sampled datasets. Additionally, we propose an ensemble model training for more effective orchestration of spatiotemporal deep-learning workloads. In this scheme, instead of training a large model which trains over the entire dataspace we train many models which train of a specific spatiotemporal extent individually. This paradigm allows, in addition to performance improvements, each model to specialize over a specific region, learning unique traits which my not be recognized with other approaches.

1.4 Expected Contributions

In this work we propose a collection of advances for storage and analytics over spatiotemporal data. In particular, our contributions include:

1. Employment of data sketching algorithms to enable in-memory analytics performance over datasets while surpass memory capacity in commodity clusters.

2. Effective dispersion-informed data replication to enable simultaneous dispersion and collocation of spatiotemporal extents in distributed environments, mitigating network I/O in data movements during analytics.

3. Implementation of solutions with popular, open-sourced communication protocols which facilitate seamless integrate into existing workflows.

4. Integration of a robust collection of spatiotemporal functionality into existing distributed analytics suites, enabling performant optimizations.

5. Ensemble based model training over spatiotemporal deep learning workloads, these offer performant orchestration paradigms while maintaining accuracy.

1.5 Experimental Setup

We executed benchmarks on one of two separate clusters. (1) 50 Hewlett-Packard DL160-G6 machines equipped with a 6-core 2.4 GHz Intel Xeon CPU E5-2620 v3 processor and 16GB RAM and (2) 50 HP-DL60-G9-E5-2620v4 machines equipped with an Intel Xeon E5-2620 (8 cores, 16 hyper-threads, 2.10GHz) and 64GB RAM. All machines are connected with a 1Gb/s ethernet switch.

1.5.1 Datasets

Solutions must support diversity in data volumes, dimensionality, and distributions. Our methodology leverages design patterns to abstract and simplify integration of diverse data collections. Each implementation of the abstraction is customized primarily to handle parsing individual formats and metadata retrieval for a specific collection. Therefore, while evaluations proposed in this work have processed the following datasets, these solutions are amenable to additional datasets and may span domains.

Feature-based

Feature-based datasets refer to data in which data consists of observations with a uniform collection of features. The number of features has serious impact on analytics performance, encompassing resource utilization, duration, and accuracy. Therefore, we have attempted to evaluate over datasets with large variances in volumes and dimensionality. Feature-based datasets used within this work are presented in Table 1.1 and are briefly described follows:

- **Million Songs:** Million Songs outlines 91 song characteristics and the year that song was recorded for, as you may guess, 1 million songs.
- **NOAA:** The National Oceanic and Atmospheric Administration (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.
- **Texas Epidemiology:** The Texas Epidemiology dataset is a high dimensional dataset containing over 2500 features per observation tracing livestock epidemics.

Table 1.1: Observational datasets evaluated in this work.

Dataset	Observations	Features	Size
Million Songs	463,715	91	425MB
NOAA	2.4 trillion	56	25TB
Texas Epidemiology	899,952	2595	26GB

Satellite Imagery

Hyperspectral imagery presents contrasting challenges with feature-based data. The variance in image formats, resolutions, spatial reference systems, etc require novel solutions to support diverse sources. Table 1.2 provides a compilation of datasets that we have evaluated. Additionally, a brief description follows:

- **MODIS:** Moderate Resolution Imaging Spectroradiometer (MODIS) captures 36 bands with spatial resolutions ranging from 250m to 1km. Images are captured using the Terra and Aqua satellites, both part of NASA's Earth Observation System. The low spatial and high temporal resolutions in this data make it particularly useful for tracking changes in the landscape over time

- **NAIP:** USDA's Farm Service Agency provides high spatial resolution (1m) imagery acquired during the growing seasons in the Continental US. These images are temporally sparse, being compiled every 3 years since 2009 and every 5 years before that.

- **NLCD:** The National Land Cover Database (NLCD) is a dataset computed by the Multi-Resolution Land Characteristics Consortium (MRLC). It provides land use classification for the United States at a 30m resolution roughly every 3 years. Currently, there are 16 classifications in additional to unclassified.

- **Sentinel-2:** Global resolution imagery is provided by the Copernicus Programme as part of the European Space Agency. Their twin satellites (i.e., Sentinel-2A and

Sentinel-2B) capture multi-spectral imagery comprising 13 visible, near-infrared, and shortwave infrared bands

Table 1.2: Descriptions of satellite imagery datasets used in this work.

Satellite	**Format**	**Spatial Reference System**	**Frequency**	**Coverage**	**Resolution**	**Band Count**
MODIS	hdf	Sinusoidal	1-2 Days	Global	200m	7
					500m	7
NAIP	zip / jp2	Transverse Mercator	2 Years	United States	1m	4
NLCD	img	Albers Equal-Area Conic	3 Years	United States	30m	1
Sentinel-2	SAFE	Transverse Mercator	5 Days	Global	10m	6
					20m	4
					60m	3

Chapter 2

Related Work

2.1 Lossy Compression and Data Sketching

Lossy compression algorithms are a class of compression which trade higher compaction rates with error-bounded data loss. These techniques were first applied as image storage formats [1, 2], where algorithms leverage similarities between neighboring pixels to achieve smaller files. It is shown that noisy images reduce the utility of these approaches [3]. Recently, these techniques have been adapted to scientific datasets, where analyses only require statistical representativeness rather than identical data. Time-series are particularly well suited, where consecutive observations are often similar. ZFP [4] aimed to ensure random read / write access to compressed data by sacrificing lossless variable length streams and instead using a fixed-length 4-dimensional data representation. ISABELLA [5] and variants [6] rearrange seemingly random, noisy data to achieve accurate model fitting. NUMARCK [7] maintains relative changes in sequential values rather than entire data. Yuan et al. [8] implements this in parallel for improved performance. Finally squeeze (SZ) [9] and variants [10] predict successive data points by linearizing multi-dimensional snapshot data. Data that is unpredictable by this method is stored in an optimized lossy compression via binary representative analysis.

Data sketches are a unique class of lossy compression algorithms that allow efficient queries over compressed data. These are particularly useful in designing approximate query processing systems. The popular Misra-Gries algorithm [11] uses a small data representation to find repeating elements in sub-linear time and space. These findings have been extended by frequency-based sketches, which represent the observed frequency distribution of data. The most common example, counting bloom filters [12], uses a fixed-size bit vector and multiple hash algorithms to bound observation frequencies. The count-sketch algorithm [13] proposes a two pass algorithm to estimate the items with the largest change in frequency between two data streams. Proposed as an alternative, the Count-Min sketch [14] maintains the linear projection of a high-dimension vector using multiple hash functions to simulate bounding, estimate vectors. Counting-Quotient filters [15] support counting and deletions while scaling out of RAM to SSD.

Tal et al. [16] have applied data sketching algorithms specifically to spatiotemporal data by proposing retrieval of summarized information on moving objects in a spatial extent. The Synopsis framework [17] comprises a collection of micro-sketches structure as leaf nodes in a forest of tries, where each trie represents a particular spatiotemporal scope and each trie level a queryable dataspace feature. Pebbles [18] applies these techniques in an order-preserving algorithm tracking consecutive data changes to represent multi-feature spatiotemporal streams.

2.2 Spatial Partitioning and Indexing

Partitioning and indexing datasets is important for efficient dataspace filtering, effective distribution, and improved resource utilization for large processing operations. Partitioning satellite imagery for efficient analysis has been proposed in a variety of domains. Recent efforts in Geographic Object-based Image Analysis (GEOBIA) [19,20] stress the importance of resolutions in rectifying inclusion of multi-source imagery. Similarly, extraction of image features for farmland [21] and irrigation [22] show accuracy variances depending on spatial bound definitions. Georganos et al. [23] process non-uniform spatial partitions to improve land-cover / land-use maps.

A variety of spatial indexing data structures have been proposed. Quadtrees [24], R-tree's [25], and many R-tree variants [26–28] build trees where each node is responsible for a spatial subset of its parent. Leaf nodes contain buckets of data and when the bucket's reach capacity the node is split, adding a layer to the tree. Quadtrees produce 4 split partitions and R-trees r split partitions. Vornoi diagrams [29] partition a plane into multiple regions (Voronoi cells) based on distances to predefined points (seeds). Voronoi diagrams provide optimizations for many spatial operations. Geocoding algorithms rely on incrementally partitioning the coordinate space in the smaller extents to produce a 1-dimensional string representing 2 dimensional spatial bounds. The greater the length (or precision) of the string, the smaller the spatial extent represented by the geocode. Geohashes are an algorithm which partitions at 32 subregions for each additional character while quadtiles (depicted in Figure 2.1) produce 4 subregions.

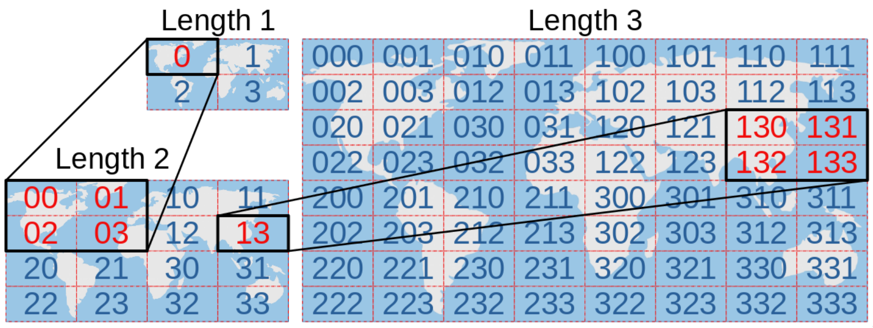


Figure 2.1: Partitioning the globe using quadtiles, each additional character produces 4 subregions.

Efforts to distribute spatial indices include the P2P R-tree [30], distributed quad-tree [31], and spatial P2P [32]. These solutions partition the indexing data structure across multiple machines, enabling more efficient and fault tolerance index lookups at scale. Scientific communities have leveraged P2P Grids [33, 34] and problem-specific storage solutions [35].

2.3 Distributed Storage

Distributed file systems are a popular storage subsystem for archival storage and distributed analytics. Perhaps the most recognizable, Hadoop Distributed File System (HDFS) [36] is the open-sourced version of the Google File System [37]. This framework partitions files into multiple blocks and distributes them over a cluster of machines, enabling parallel processing and fault tolerance. HBase [38] and Hive [39] are applications built on HDFS to expose a relational database interface. Lustre [40] and Gluster File System [41] aim to provide highly scalable solutions, sacrificing a homogeneous deployment infrastructure for strict partitioning of the data and control planes which is useful in HPC environments.

Recent work has extended the HDFS codebase to make it amenable to in-memory storage, facilitating more efficient file retrieval. Triple-H [42] integrates a data hierarchy paradigm, where data transitions between different layers of the storage subsystem (i.e., RAM, SSD, HDD) to ensure popular data is memory-resident. This allows the system to cope with much datasets much larger than the collective memory capacity of the cluster. Tachyon [43] relies on data lineages and data check-pointing 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.

Native spatial support has been introduced into the HDFS ecosystem. Parallel-secondo [44] is a parallel spatial DBMS using Hadoop for spatial support in the analytics layer. VegaGiStore [45], HadoopGIS [46], and SpatialHadoop [47] are extensions to the Hadoop source-code to support spatial indexing and analytics within the framework. These typically rely on proven spatial indexing paradigms (e.x., QuadTrees, R-Tress, etc) which require an inexpensive preprocessing phase during data ingestion. Similarly, H-Base, the relational database built on HDFS, has seen extensions including MD-HBase [48], H-Grid [49], and HBase-Spatial [50] which provide spatial support within the framework.

Additionally, modern relational and NoSQL database systems have extensions supporting spatial queries. [51] and [52] provide spatial query support for points, lines and polygons for MariaDB and Postgres respectively. Geomesa [53] combines three elements of geometry and time using a custom geohash implementation to provide queries using Google Cloud Bigtable [54], Apache HBase [55], Cassandra [56], and more.

2.4 Distributed Analytics

The Apache Spark [57] framework provides efficient distributed, in-memory analytics [58]. The framework's crux is Resilient Distributed Datasets (RDDs) which offer an abstraction over structured, in-memory, distributed data allowing for simple transformations and manipulations. SparkSQL [59] is an extension enabling SQL-like operations over traditional RDDs. Spark has been extended to provide support for relational data processing [59]. This simplifies the Spark interface, easing adoption into workflows.

A variety of efforts target spatial functionality within the Hadoop MapReduce framework. Individual spatial functions have been proposed addressing range queries [60], K-nearest neighbors [61], and all-nearest neighbors [62]. Join operations, focused on spatial parameters and K-nearest neighbors, were evaluated in [63], [64], and [65]. Full spatial functionality suites have been proposed using Voronoi-based spatial MapReduce operations [66], distributed spatial indices [67], and at the Hadoop language interface layer in Pigeon [68], an extension of Hadoop's PigLatin [69] language.

Spatial functionality has been integrated into the Apache Spark ecosystem in SparkGIS [70], GeoSpark [71], LocationSpark [72], and Simba [73]. These systems rely on dynamically building a spatial index over the datasets during data source reads. This results in two limitations (1) the source dataset needs to fit in-memory and (2) dynamic indexing of input data reduces the viability of ad-hoc queries on variable datasets. They extend, in varying capacity, Sparks User-Defined Type abstraction to provide first class support for spatial objects and manipulations.

2.5 Distributed Model Training

Training accurate models over large, distributed datasets introduces complex challenges in coordination and resource utilization. A broad spectrum of solutions have been proposed to with the focus on reducing training times without sacrificing accuracy. Sampling is a commonly used strategy for dealing with voluminous datasets. Different strategies [74–76] have been proposed to create a smaller representative subset of the original dataset. Although sampling enables analysis of voluminous data, it becomes infeasible for optimizing problems with a large number of parameters. Techniques that enable full use of data that is large in volume, diversity, and dimensionality facilitates a deeper understanding [77]. Agarwal et al. [78] have shown that increasing the sampling size of such data usually leads to better accuracy. To complement data sampling, generating representative synthetic datasets based on original data has been explored in statistical work. MUNGE [79] and many SMOTE variants [80, 81] 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.

Orchestrating deep learning workloads, to train a single model on multiple machines, offers opportunities to reduce training times with increased parallelism [82]. The two competing approaches are data parallelism [83–86] and model parallelism [87]. Data parallelism trains a copy of the entire model on each machine and synchronizes model weights after each epoch. This is the primary technique used in Tensorflow [88] and PyTorch [89] distributed packages. It introduces significant network overheads, which may transform network I/O into a bottleneck for large models. The Ring AllReduce algorithm [90] combats this by promising optimal bandwidth usage as long as network buffers are large enough. Alternatively, model parallelism distributed parts of the model across the cluster. The necessary training consensus among hosts mean this solution often suffers from concurrency-related training issues [91]. Consequently, it's only applied on very large models [92]. Orthogonal to data and model parallelism, some approaches partition the problem using modular reinforcement learning [93–95] where a complicated task is decomposed into isolated sub-tasks.

Spatiotemporal data offers opportunities for domain specific optimization. Bruhwiler et al. [96] uses auto-encoders to convert satellite imagery into succinct embeddings, facilitating efficient training with data size reductions. Khandelwal et al. [97] uses nonblocking I/O to generate satellite imagery imputations over satellite imagery at myriad spatiotemporal scopes.

Implementations of the aforementioned distributed deep learning orchestration techniques have been included in multiple frameworks. Extensions to the Hadoop MapReduce framework [58,78,98,99] propose optimizations targeting complex model training tasks. [100,101] employ a graph abstraction to express complex computations, simplifying the distributed evaluation. Broader approaches to machine learning have been addressed in [102–105], where systems support more generalized algorithms aiming to be a one size fits all solution.

2.6 Discussion

The aforementioned work present untold innovation, consistently pushing the current state of the art. With a few more intricate pieces we believe they can be leveraged to combat modern challenges in the storage and processing of spatiotemporal data at scale. Current distributed approaches are typically implemented with generalized functionality, meaning they may operate over diverse datasets in diverse environments. While this is often a advantageous property, we see glaring inefficiencies when processing domain-specific data. Spatiotemporal data, containing both a time and space component, is uniquely positioned for more efficient indexing, facilitating faster data retrieval and more effective analytics scheduling. Current solutions offering spatiotemporal support have been implemented as a afterthought, layering functionality on top of existing protocols which restrict performance. In this dissertation we purport to introduce first-class support for spatiotemporal support in distributed storage and analytics frameworks; and in doing so, improve the accuracy and performance of large-scale evaluations.

Chapter 3

Partitioning Spatiotemporal Data

As dataset volumes increase so does the cost of processing. This is observed not only in the obvious linear relationship with data sizes, but also in in dataset filtering operations where index lookups are less efficient. These expenses compound by the intrinsic nature of spatiotemporal analytics, where operations are iteratively defined and data tends to span statically defined boundaries. Additionally, analytic operations over a specific spatiotemporal extent are often processed in batches, where large batch sizes often exceed memory capacity of commodity clusters. This construct is outlined in Figure 3.1 where data sizes of unpartitioned Sentinel-2 images are multiple orders of magnitude too large to fit in GPU memory. The right-most 3 lines represent images greater than 1000 by 1000 pixels at varying resolutions, even with batch sizes of 2, which is not advised, they exceed 8GB of memory consumption.

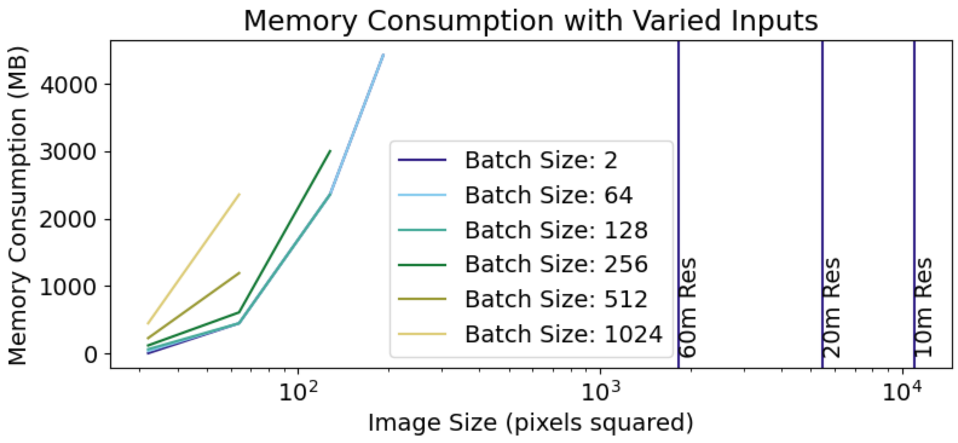


Figure 3.1: GPU training memory requirements in logarithmic scale, undivided images are 100 times larger than memory capacity.

We have developed staging solutions which support a variety of observational datasets. Our objective is to partition the dataspace to facilitate effective filtering over both spatiotemporal attributes and data features. Specifically, we focus on three approaches.

1. Applying sketching algorithms to significantly reduce multi-dimensional dataset sizes and enable filtering on a variety of features.

2. Partitioning feature-based data into blocks where data is spatiotemporally contiguous, promoting targeted, sequential disk access for each specific spatiotemporal extent.

3. Reconciling diversity in hyperspectral image formats, resolutions, band definitions, and spatial reference systems to effectively process satellite imagery from an assortment of sources.

3.1 Sketching Datasets

Sketching datasets results in significant reductions in dataset sizes, typically measurable in orders of magnitude, and efficient dataset indexing structures. The Fennel [106] sketch focal point is *Discretized Feature Vectors (DFVs)*. For each dataset feature we dynamically compute discretization boundaries based on the online kernel density estimation (oKDE). Using these boundaries we convert each observation into a DFV, or a vector of discretized features. Fennel then tracks frequencies of each unique DFV, further reducing dataset sizes. Specifically, our methodology for productively sketching datasets involves two stages:

1. The groundwork phase focuses on distributed calculation, and subsequent exchange, of global statistical properties of the feature space.

2. We then sketch on-disk data to ensure effective compactions while preserving representativeness of the feature space. Portions of the sketch are shuffled to conserve memory footprints and fast query evaluations.

3.1.1 Groundwork Phase

The groundwork phase constructs statistical properties of the feature space and uses online kernel density estimation functions to compute feature discretization boundaries. We designed a MapReduce job that does a single pass over the distributed dataset in the Map phase, and a computationally inexpensive Reduce phase.

We compute feature discretization boundaries based on a data sample of around 50,000 observations. We identify a reservoir sample at each host with a proportional number of observations relative to the entire dataset. Reservoir sampling is an online sampling technique, which ensures that each observation has an equal probability of being present in the final sample. We then combine the samples from each host and compute an online kernel density estimation (oKDE) function for each feature. We iteratively increase the number of bins until the normalized root means squared error (NRMSE) between the discretized values and the sample is below a configurable, predefined threshold (0.025 in our experiments). During each iteration, discretized boundary values are computed to split the oKDE into bins containing equal observation frequencies. Tuning the NRMSE threshold is trade-off between dataset compaction rates and the statistical representativeness. The result of this operation is a list of bounds that split the feature space into bins with equal observation frequencies

During the Map phase we iteratively track statistical properties of the feature space using Welford's online method evaluated over each observation. The Reduce phase leverages the well-defined algorithm to merge separate instances of Welford's from each node, resulting in statistics including minimum, maximum, mean, and average values for each feature in the data.

3.1.2 Supported Multiple Sketches Over the Same Data

We initialize sketches in Fennel by defining a set of active features and their corresponding discretization boundary values. This enables defining multiple sketches over the same dataset encompassing any subset of features and producing myriad compaction rates and levels of synthetic data accuracy.

We store unique DFVs and their accompanying frequencies. Employing a hash map, as opposed to a tree structure, reduces overheads. During query evaluations, a tree-based organization requires traversal of many levels; data with high dimensionality presents a combinatorially explosive tree (e.g.; 2000 features, with 30 discretized values each, results in 30^{2000} possible leaf nodes). Query wildcards where some features are unconstrained, results in an exponential increase in the number of tree path traversals.

Microbenchmark: Figure 3.2 depicts the compaction effectiveness of Fennel over the NOAA dataset. We iteratively loaded each month and noted sketch memory consumption. We see the cumulative sketches compaction rates improve from the first month to the last increasing from 10x—13x, 25x—27x, 250x—280x, and 15,000x—87,000x for 56, 28, 14, and 7 features respectively. Of note (1) reducing the number of features has a significant impact on sketched data size and (2) sketched size follows a logarithmic trend where compaction rates improve as the data volumes increase.

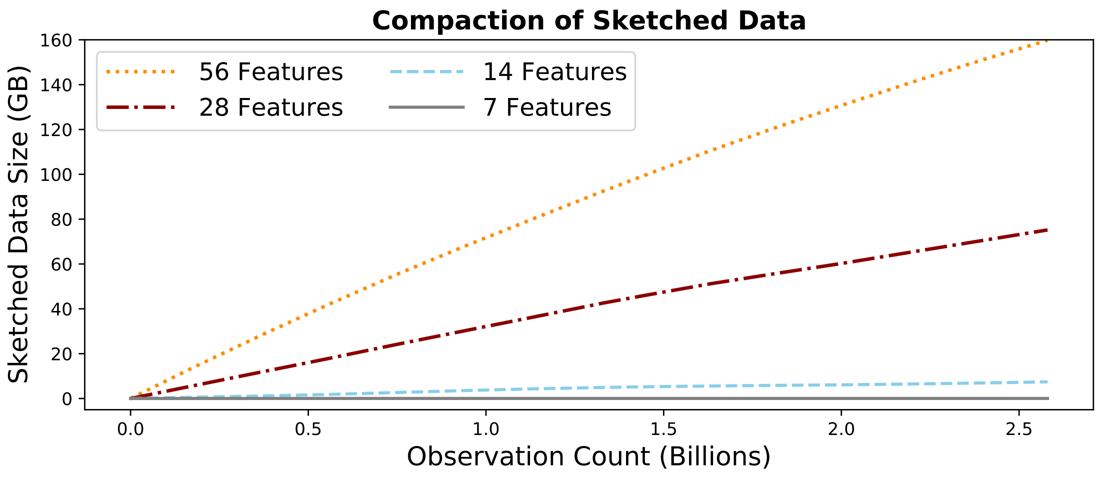


Figure 3.2: Dataset compaction using the Fennel sketch.

3.2 Spatiotemporal Block Indexing

Sequential file reads offer the best performance, in terms of throughput and transfer times, during data retrieval. However, this process lacks support for spatiotemporal filtering, where identification and retrieval of a specific spatiotemporal scope requires iteration over the entire dataset and filtering of observations individually. This introduces a number of inefficiencies including computational overhead and unnecessary disk and network I/O.

The main challenge in identification and retrieval of specific spatiotemporal scopes is that data reporting seldom aligns with spatiotemporal attributes. Rather, files tend to contain a diverse collection of spatiotemporal scopes. This contributes to two inefficiencies. First, relying on filtering at the observation granularity incurs unnecessary computational overhead, disk I/O, and network I/O. Second, non-sequential disk reads entail significant disk head movements, negatively impacting read performance

Atlas [107] relies on chunking the dataset to aid in efficient data access. This is performed by partitioning input data files into multiple blocks which may then be distributed and replicated over the cluster. As blocks are written to the system, observations are dynamically rearranged to align with spatiotemporal boundaries. We refer to this segmentation as producing *striped sets*. The process begins by computing the geohashes for each observation, an operation that supports points, lines, and polygons. Next, observations are rearranged so that spatial scopes are contiguous, and temporally adjacent and increasing, within the block. This procedure is performed in-memory to reduce computational and I/O overhead.

Rearranging observations within blocks to align with spatiotemporal boundaries simplifies identification and retrieval of dataspace subsets. The shuffling allows for fast identification of a spatiotemporal dataspace, where positional indices (i.e. data offsets and length) for each block correspond to a spatiotemporal range. Additionally, data retrievals can be performed with sequential disk reads, mitigating the performance degradation caused by excessive disk head movements. The process of chunking and indexing the dataspace is outlined in Figure 3.3.

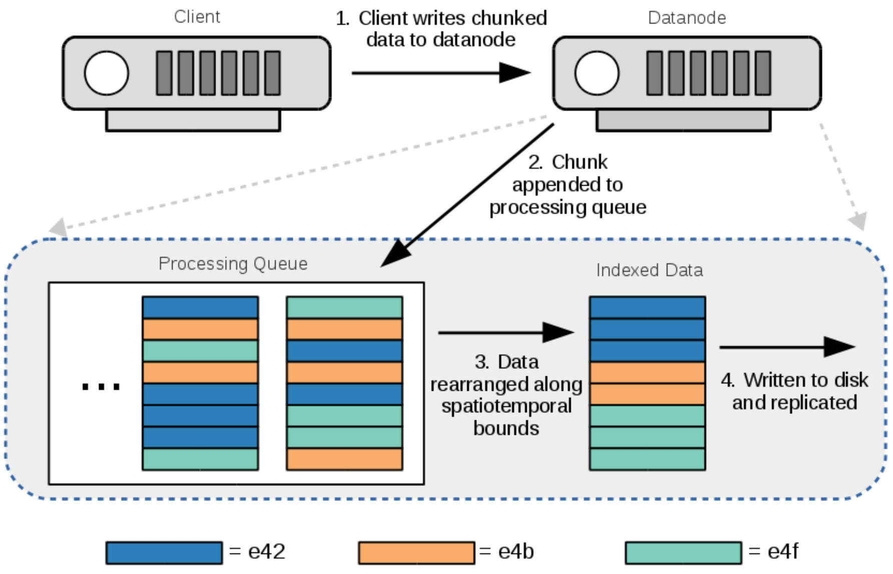

Client
1. Client writes chunked data to datanode
Datanode
2. Chunk appended to processing queue
Processing Queue
...
3. Data rearranged along spatiotemporal bounds
Indexed Data
4. Written to disk and replicated
= e42
= e4b
= e4f

Figure 3.3: Spatiotemporal block indexing within Atlas framework.

Microbenchmark: In Figure 3.4 we profile the block indexing performance when writing 1TB of data into an Atlas cluster of 50 hosts. We performed three separate experiments with three different block sizes, namely 64MB, 128MB, and 256MB. Block index durations were sorted from longest to shortest and plotted along the x-axis. We see that index duration is consistently efficient, on the order of a few seconds. Moreover, data insertions in Atlas are asynchronous, where completing the indexing over one block does not delay insertion of the next. Additionally, total data indexing time does not significantly differ across the three experiments.

3.3 Reconciling Differences in Image Attributes

Hyperspectral imagery introduces novel challenges in partitioning, as temporally consecutive images are seldom constrained by identical spatial regions. Additionally, pro-

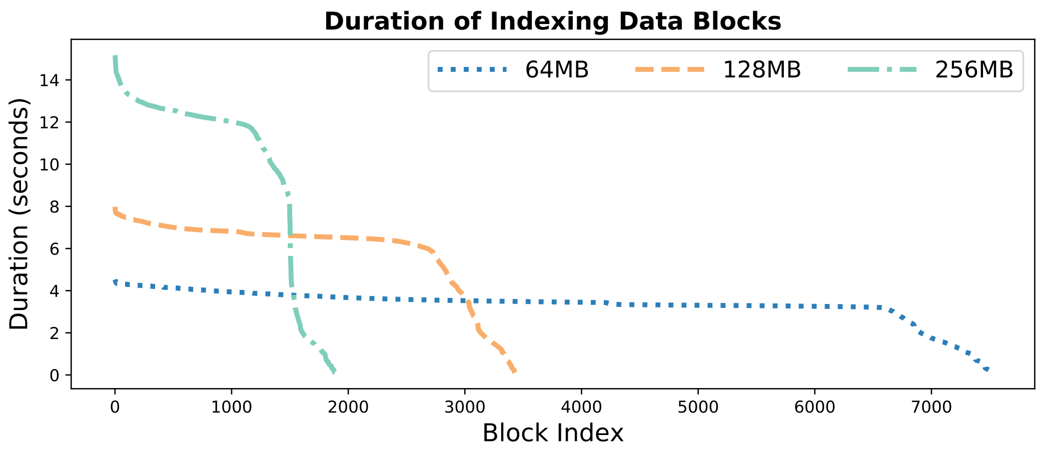


Figure 3.4: Atlas block indexing duration.

cessing images from diverse sources requires remedies for image formats, pixel resolutions, and different spatial reference systems among other attributes.

Our methodology encompasses steps to ensure discovery, load balancing, timeliness, and throughput of operations. We harness two distinct spatial partitioning schemes to support the desired objectives: geohashes and quadtiles. The choice of the algorithm is mostly influenced by the input dataset and types of analyses. The spatial reference system within the imagery also plays a role in this choice: geohashes work with `[latitude, longitude]` bounding boxes while quad-tiles leverage Mercator projections. Another consideration is the granularity of the hierarchical splits as the length of the geocode increases. Geohashes produce 32 subregions at each level, while quad-tiles result in 4 subregions.

As each hyperspectral image comprising multiple bands is being ingested, we extract metadata from the images and organize them so that they are amenable to query evaluations. Our dispersion scheme allows us to load balance storage, query evaluation, and model training workloads. Once the spatial granularity is finalized, the geocode and the DHT node responsible can be calculated deterministically and without coordination. Furthermore, since all data for a particular geocode are stored on the same DHT node our methodology ensures collocation of data from spatial regions across different temporal ranges on the same machine. A consequence is that our methodology supports targeted processing of fine-grained spatiotemporal extents. Model training in distributed environments often entail substantial data movements as data are pulled from multiple nodes. Since we ensure collocation when training models for a particular spatiotemporal extent, we substantially alleviate the adverse impact of network I/O during data transfers.

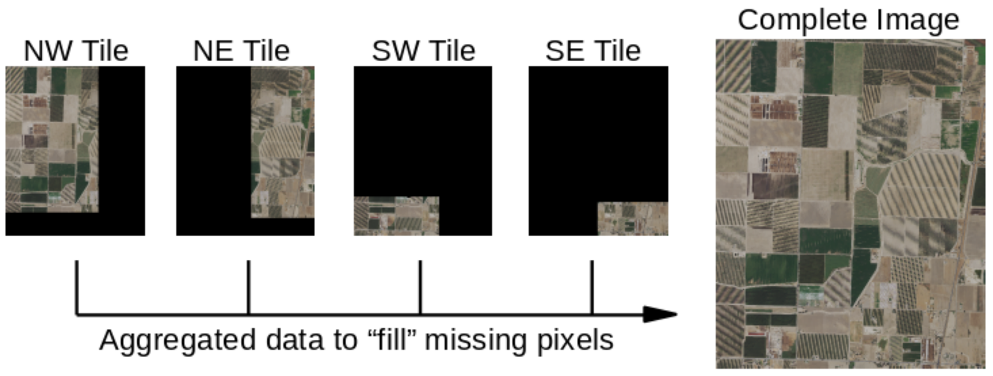


Figure 3.5: An example of the image 'fill' procedure.

Our spatial partitioning scheme may result in images where portions of the geocode bounded image are completed with "fill value" pixels. This is because input imagery seldom aligns exactly with geocode boundaries. Alternatively, cloud coverage results in similar occlusions. To remedy, we provide the capability to combine temporally-proximate data for the same spatial region to construct a synthetic image that satisfies maximum thresholds for pixel or cloud coverage proportions. We are able to efficiently identify collections of images that share a spatiotemporal scope and for which the pixel / cloud coverage is over the specified threshold. By combining portions from these different hyperspectral images we are able to dynamically splice together an image with the desired coverage. This construct is depicted in Figure 3.5, where four partial images from the same spatiotemporal extent that are aggregated to produce a complete image. These temporal reconciliations of the dataspace ensure a complete dataset and are useful for effective deep learning over satellite imagery.

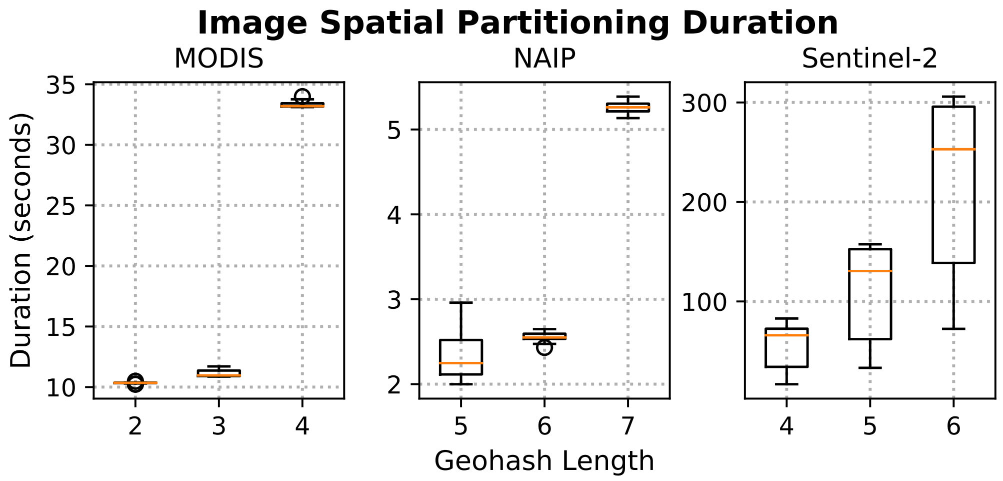


Figure 3.6: Spatiotemporal image partitioning duration.

Microbenchmark: In Figure 3.6 we performed a series of experiments profiling raw-image processing, including spatial partitioning and distribution. We chose 3 unique geohash lengths for each dataset based on image pixel resolutions – with all these datasets there is a point at which image resolutions become too small and the utility diminished. The first observation is the relative difference in processing time between datasets. This may be attributed to the difference in the number of bands and unique resolutions within each image. Next, we see a difference in variance among processing times between datasets. We noticed a strong correlation between file sizes and processing time. MODIS and NAIP images are typically uniform across observations. Sentinel-2's higher bands and varying resolution introduce variances. Consequently, decompression, parsing, and data transfer speeds are higher for Sentinel-2.

Chapter 4

Distributed Storage

Modern data scales guarantee that storage and analytics will overwhelm multi-faceted resource availability on a single machine. Therefore, common solutions rely on distributing data over a cluster of machines. There are inherit challenges in this approach. Foremost of which, particular analytics tasks tend to perform well with a specific data distribution scheme. At the extremes, dispersion focuses on distributing data evenly across each machine which is useful for parallel processing; alternatively, locality supports processing the entire dataspace on a single host which is useful when algorithms require an omniscient view (e.x., data joins). Unfortunately, distribution schemes tend to conflict (i.e., dispersion vs. locality) and there is no one size fits all solution. Therefore, it is extremely difficult to support diverse analytic tasks without incurring data movements, in which are associated with expensive disk I/O (during reads and writes) and network I/O (for inter-machine transfers).

We purport to develop distribution schemes, agnostic of data format, which may simultaneously provide both dispersion and collocation of spatiotemporal extents. To be effective these must be decentralized, deterministic, and preserve dispersion and collocation properties. Given the iterative nature in the analytical process, the impact of these inefficiencies are amplified. Specifically, we address these challenges by:

1. Leveraging the properties of sketched data to dynamically distribute observations across a cluster with minimal resource costs.

2. Employ distributed hash tables (DHTs) to effectively load-balance data distribution based on spatiotemporal properties.

3. Inform dataspace replication with dispersion properties to facilitate simultaneous dispersion and collocation of spatiotemporal extents.

4.1 Processing Sketched Data

At the lowest granularity, sketched data manifests as Discretized Feature Vectors (DFVs), which are representative to a collection of observations. Dataspace queries over distributed sketched data return a collection of DFVs (explored further in Section 5.1). The massive dataset size reductions inherit in sketched data facilitate dynamic rearrangement of sketches to process exploratory datasets under a variety of distribution schemes, enabling more efficient processing. Once a subset of the dataspace has been identified, the next challenge is to distribute sketched data over a cluster of hosts while load balancing the expected processing being performed.

The crux of sketch distribution involves tracking real-time resource utilization statistics across the set of available machines and computing data sizes to inform distribution. Using this utilization information, we are able to frame this process as a combinatorial optimization problem where we distribute data such that nodes have proportionally equal resource utilization; this is equivalent to ensuring that each node holds roughly the same number of observations. Given that each observation is subject to identical processing, we expect the loads to be balanced. This problem mimics the classic knapsack problem.

Based on efficient multi-objective solutions for the knapsack problem [108] and resource allocation [109] 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.x., many small shards vs. few large shards).

Mircobenchmark: In Figure 4.1 we evaluate performance of genetic algorithms against simple greedy and round-robin shard distribution techniques. The x-axis is displayed as "# of machines - # of shards". 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 4.1 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.

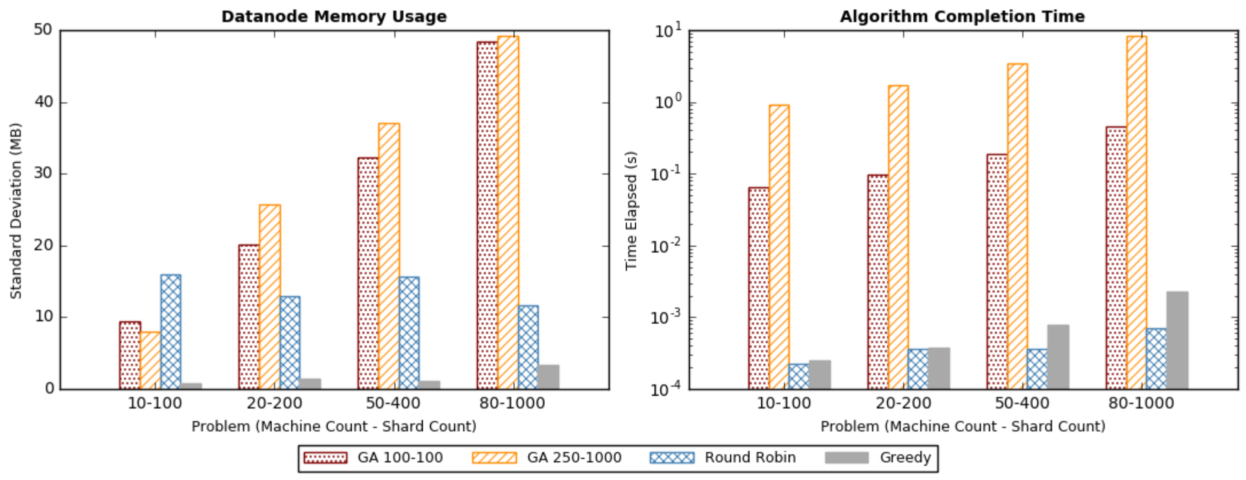


Figure 4.1: Evaluation of data sketch distribution techniques.

4.2 Distributed Hash Tables

Distribute hash tables (DHTs) are commonly employed in distributed storage frameworks for their deterministic and decentralization properties. Each cluster node is assigned a collection of tokens in a specific hash range. Gossip between nodes ensures each node maintains the mapping between tokens and cluster nodes. During data distribution, each node individually evaluates a hash function over processed observations. Using on the resulting value, the node consults the token to cluster node mapping to determine which cluster node is responsible for the data.

We use a multi-token distributed hash table (DHT) to disseminate geocode-based spatiotemporal data throughout the cluster. We support dispersion based on any substring length of the geocode. For example, processing length k ensures that all images beginning with the same k characters are collocated. Alternatively, using geocode length k-1 disperses images so that data from spatially proximate subregions are collocated. The geocodes are passed through a cryptographic hashing function (SHA-1) and each DHT node is responsible for a contiguous portion of the cryptographic hash space. The dispersion properties of the cryptographic hash function ensure that the load is uniformly dispersed over the cluster. In particular, this allows for fine-grained and dynamic load balancing of storage workloads within the system. This, in turn, ensures that subsequent pre-processing and model training workloads are dispersed and dynamically balanced as well.

Table 4.1: Image partition distribution metrics.

Metric	Mean	Standard Deviation
Dataset Size	139.5GB	1.5GB
Image Count	255k	2.8k

Microbenchmark: To highlight the effectiveness of our DHT we staged a satellite data collection and report on the distribution of image geocode splits. In this experiment, our base dataset comprises all available Sentinel-2 images for the Continental United States during July 2019, roughly 6,400 images and we operated in a 50 node cluster. We staged this collection by partitioning the data using geohashes of length 5, producing bounding regions approximately 5km x 5km. Table 4.1 profiles the effectiveness of our distribution scheme using a variety of metrics. We present dataset size, or the aggregate size of all images at a specific node, and image count, which is the total number of images at a specific node. As can be seen, the standard deviation are quite small for each metric indicating balanced data distribution.

4.3 Dispersion-Informed Data Replication

Supporting a data distribution technique that performs well in myriad analytic flavors is difficult. Many analytics tasks benefit from specific data distribution policies – a one size fits all solution does not exist. This problem is compounded by support for spatiotemporal data, where analytics are often performed on dataspace subsets, meaning control of data distribution must be performed at a much finer granularity. Atlas attempts to balance the trade-off between distributing data to provide efficient analytics and load-balancing data storage. Skewed data distributions result in host hot-spots, mitigating many of the advantages of distributed storage and processing.

The Atlas distribution algorithm purports to support both data dispersion and locality simultaneously. Clients begin by writing data to the cluster. Initial block placement is based on load-balancing disk utilization among cluster hosts. The storage node dynamically computes spatiotemporal scopes within data blocks as they are written (as described in Section 3.2).

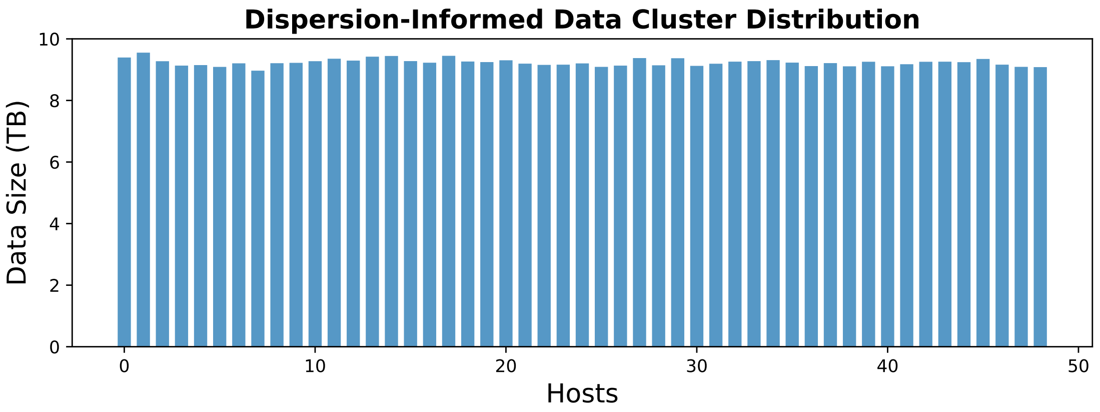


Figure 4.2: Data block cluster distribution within Atlas.

Then each block is asynchronously replicated to multiple hosts (the replication level is configurable with a default of 3). The replication policy for data within a spatiotemporal subspace favors hosts that have stored similar data. This process is assisted by the Atlas namenode, which contains a spatiotemporal index over the data (as discussed further in Section 5.2). Thus, the initial data write is processed on a pseudo-random host, and replicas are written to hosts containing data from a similar spatiotemporal extant.

Atlas' data distribution policy results in data for each spatiotemporal scope that is thinly distributed over the cluster hosts, where replicas are concentrated on a small subset of hosts. This provides the ability to access spatiotemporal subsets with data dispersion and/or locality principles, facilitating analytics scheduling with host data locality that is aligned with the particular task.

Microbenchmark: In Figure 4.2 we profile data distribution across Atlas hosts. For this experiment we load 150TB of data into the cluster using a 128MB block size. Each bar on the x-axis corresponds to a datanode and the y-axis displays the cumulative data storage size at that particular host. Overall, the experiment highlights Atlas' ability to load-balance data storage while preserving the trade-off between data dispersion and locality for individual spatiotemporal scopes, comparing and contrasting different regions and/or durations.

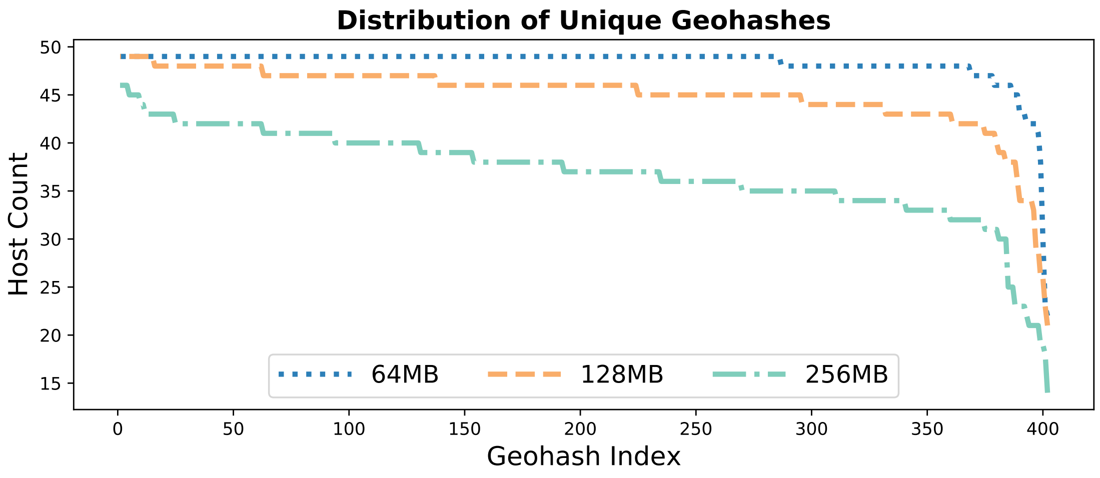


Figure 4.3: Number of hosts for each unique geohash in Atlas distribution.

Microbenchmark: Figure 4.3 highlights the effectiveness of Atlas' thin geohash distributed over cluster hosts. For this experiment we loaded 1TB of data into a cluster of 50 machines (1 namenode and 49 datanodes). We identified the collection of unique indexed geohashes, sorted the number of hosts each geohash was distributed over, and plotted the results across the x-axis. We performed this experiment for block sizes of 64MB, 128MB, and 256MB. We see that smaller block sizes provide a wider distribution, where each unique geohash is distributed over more hosts. However, even geohashes in the 88th percentile for 256MB blocks are dispersed across 35 hosts, enabling efficient parallel processing for the particular geohash.

Chapter 5

Performant Data Retrieval

Reduction operations are important for data processing tasks to identify and retrieve subsets of the dataspace. This is particularly important for spatiotemporal analytics, which commonly evaluate over a particular spatial and temporal range. To mitigate resource inefficiencies, filtering approaches need to leverage dispersion properties to ensure decentralized, deterministic query evaluation. This means queries will only contact one, or a small number, of cluster nodes. Additionally, data retrieval at each node must be targeted and sequential, mitigating overheads in processing unnecessary data.

Distributed File System Integration: There is a rich ecosystem of analytical engines that support efficient, distributed analytics. Interfacing with these is paramount for integrating our work into existing workflows. The Hadoop Distributed File System (HDFS) interface is a mature, well-defined specification that is leveraged by several analytical engines such as Hadoop, Spark, TensorFlow, Flink, etc. for hosting datasets (inputs) or writing results (outputs). Therefore, a primary objective of our work is to design our systems with adherence to HDFS's communication protocols. However, note that these solutions serve as an example for interfacing with a popular distributed file system. Though we have chosen HDFS, they may be applicable in a variety of environments.

In HDFS, the interprocess RPC-based communication is based on Protobuf [110], 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.

Architecturally, HDFS employs separate namenode and datanode applications. Control plane communication is isolated in the namenode and leverages the aforementioned custom RPC implementation. We implement 16 different operations for client to namenode communication and 3 operations for datanode to namenode communication. A few example operations are provided in Table 5.1.

Table 5.1: Example of control plane messages 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).

5.1 Enumerating Sketches

A first step in effectively retrieving data from sketches is identifying portions of the data space that are of interest. The sketch supports relational and statistical query evaluations, leveraging small internal tree-based indexing data structures over the discretized boundaries. The results of these query evaluations identify specific DFVs that satisfy the specified query constraints. They manifest as a collection of tree paths, each of which represents a sliver of the spatiotemporal scope encapsulated by the overall sketch.

We organize results from these query evaluations in our *Slice* data structure. The Slice organizes information as a forest of trees, where each level represents a feature and the dataspace and leaf nodes contain observation frequencies and summary statistics. 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. 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.

5.1.1 Materializing Slices

Each tree path within the Slice represents a spatiotemporal scope and maintains summary statistics for each dataspace feature the number of observations reported for uniquely identifying path. 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.

Microbenchmark: In Table 5.2, 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.

Microbenchmark: In Table 5.2 we profile the statistical similarity for between 2 features of each dataset comparing synthetic and original feature values. 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 5.2 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.

Table 5.2: Accuracy of synthetic observations generated from sketched data.

Dataset *(Observations)*	**Feature** *(Unit)*	**Mean**		**Standard Deviation**		**Kruskal-Wallis P-Value**
		Original	**Synthetic**	**Original**	**Synthetic**	
NOAA *(720)*	Temperature *(Kelvin)*	288.155	288.183	6.514	6.533	0.961
	Humidity *(Percent)*	77.253	77.152	19.488	19.561	0.957
NOAA *(85920)*	Wind Gust *(Meters / Sec)*	4.054	4.054	2.636	2.637	0.949
	Pressure Surface *(Pascal)*	82280.923	82279.088	5558.614	5562.483	0.984
NOAA *(515640)*	Pressure Tropopause *(Pascal)*	20189.206	20189.755	1146.239	1146.937	0.922
	Precipitable Water *(Millimeters)*	16.949	16.948	3.499	3.501	0.985

5.1.2 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 [111] 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 5.3 displays examples of the diverse set of encoding formats that we support.

Microbenchmark: In Figure 5.1 we present the duration for generation of synthetic feature values based on Slice statistics in a variety of formats. We see formats have significant impact on generation times, where binary formats have increased performance when compared with textural data.

Table 5.3: Example of available sketched data materialization formats.

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

5.1.3 Instrumentation of Slices

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.

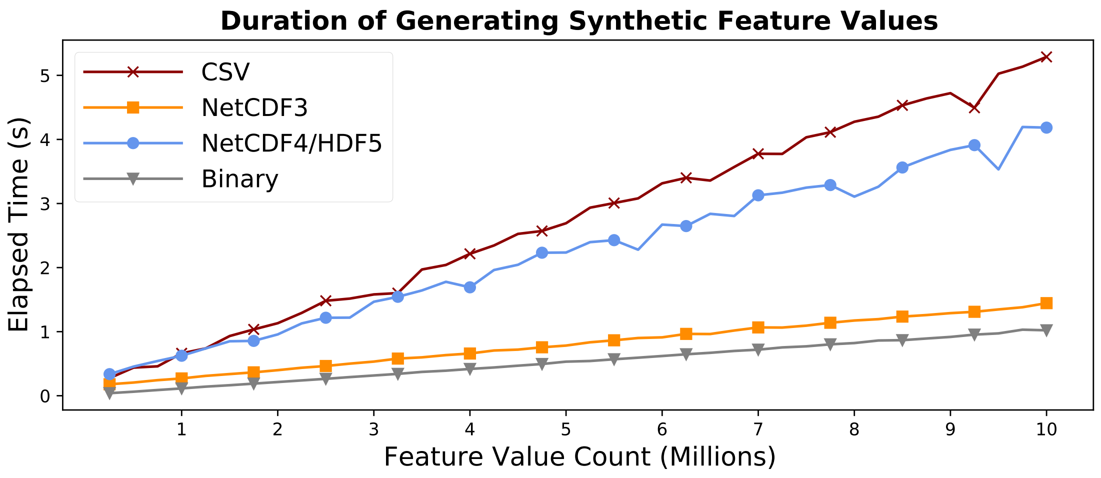


Figure 5.1: Duration to generation synthetic observations from sketched data in myriad formats.

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.

We have developed a Java API to drive the instrumentation of slices. We provide simple fluent-style interfaces for initializing Slices, chaining of instrumentation operators for Slice configuration and refinement, and robust Slice materialization. The Scala language is based on the Java Virtual Machine (JVM). Additionally it provides an interactive shell, enabling dynamic Slice configuration. 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.

In Listing 6.1, we present a short Scala example highlighting the functionality of our Java API. In lines 2 - 5, we initialize a Slice using a 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 involves management of the exploratory dataset distribution over cluster nodes.

Listing 5.1: Scala code outlining Slice Java API functionality.

```
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)
```

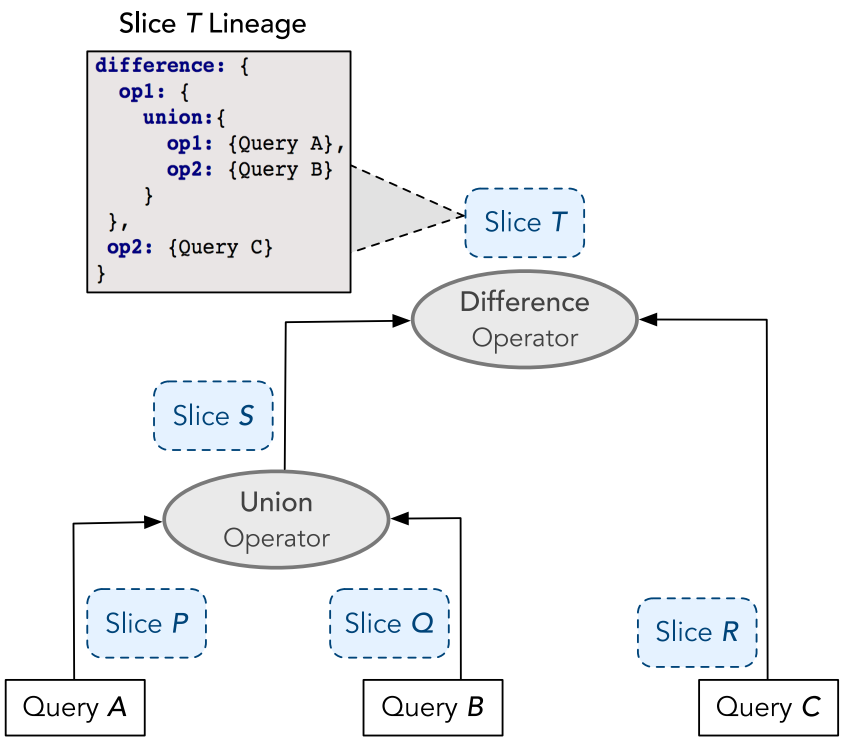
```
    .selectRange("temperature", 260.0f, 300.0f);

// slice materialization
Context.anamnesisMaterializer("10.0.0.40", 8020)
    .setShardInterval("temperature", 250, 10)
    .setFeatureIndexes(Array(2, 1))
    .setFilename("/user/hamersaw/slice1.csv")
    .setInflationRate(0.8)
    .execute(slice3);
```

5.1.4 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 5.2. 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. 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.



```
difference: {
  op1: {
    union:{
      op1: {Query A},
      op2: {Query B}
    }
 },
 op2: {Query C}
}
```

Figure 5.2: 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.

5.1.5 Integration With Analytics Engines

We provide seamless integration with analytics engines by presenting an HDFS-compliant interface over our distributed sketches. Synthetically generated observations are organized into data blocks. Our communication protocol implementations are identical to canonical HDFS serving as a *drop-in* replacement. We have extensively tested 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.

We compared Anamnesis [112] 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 Slice materialization scheme allows 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 re-materializing affected data, whereas corruptions of the underlying sketch require Slice redistribution. 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 [43] 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.

We present our dataset filtering analyses in Figure 5.3. 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.

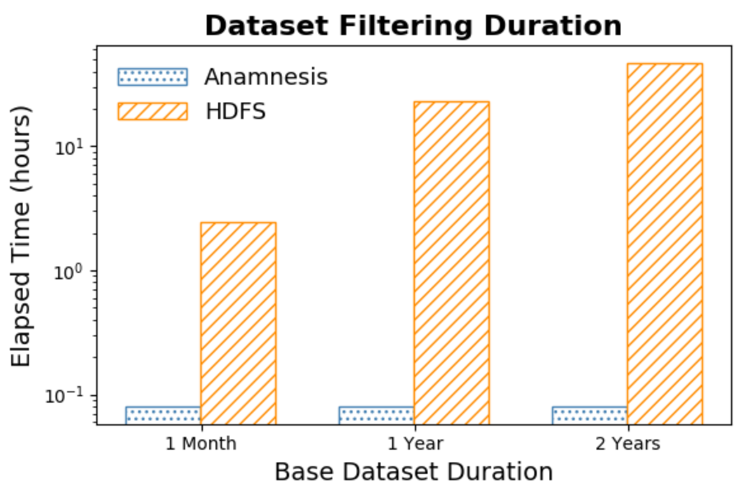
We explore filtering operations duration in Figure 5.3a. Anamnesis demarcates the times for the Synopsis query and data distribution. We see a 30x, 288x, and 587x reduction in filtering duration when compared to HDFS with base datasets of 1 month, 1 year, and 2 years respectively. Most importantly, the duration to filter using Anamnesis remains constant (approximately 5 minutes) and is independent of the dataset size.

In Figure 5.3b 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.

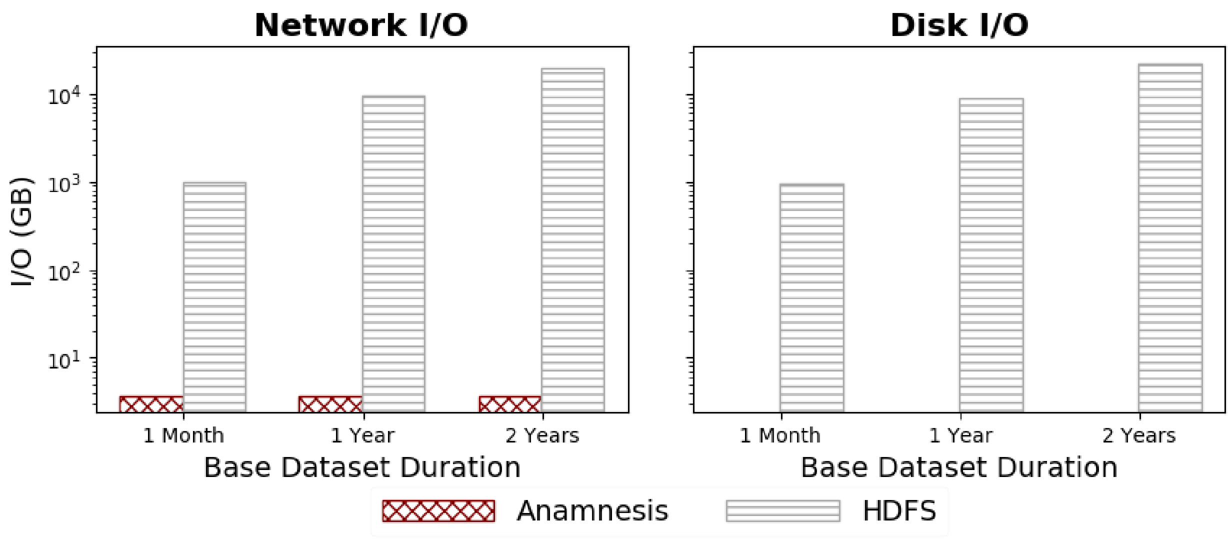
In all experiments, filtering with HDFS requires the entire dataset to be read from disk. This results in approximately 900GB of disk I/O to filter 1 month and just over 9TB in the case of 1 year. Alternatively, by leveraging the Synopsis sketch Anamnesis performs all filtering actions in memory, resulting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

5.1.6 Alleviating Memory Contention

Our solution for presenting sketched data with HDFS-compliance relies on in-memory working datasets and is therefore subject to memory contention constraints. The chal-



(a) Duration of filtering operations.

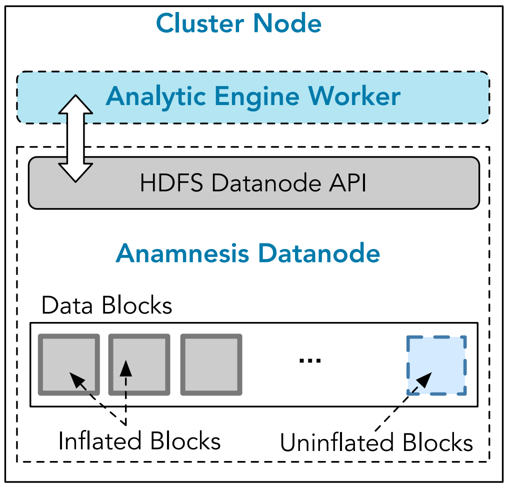


(b) Network and disk I/O incurring during filtering operations.

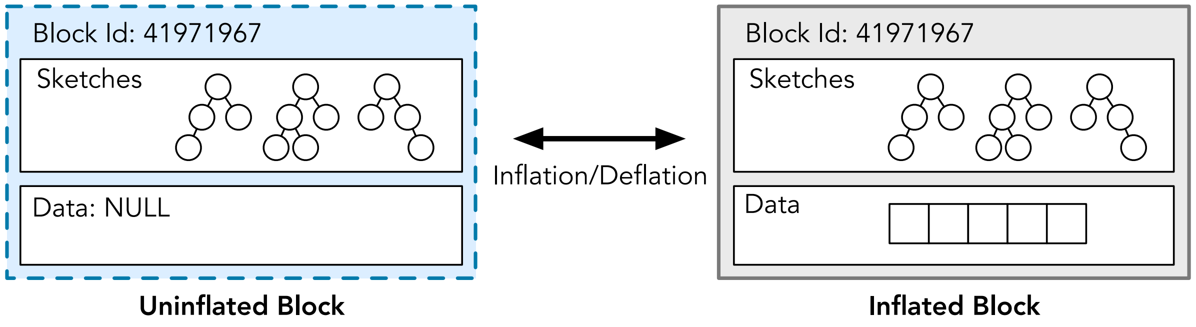
Figure 5.3: Dataset filtering metrics comparing Anamnesis with HDFS using Apache Spark.

lenge 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



(a) Depiction of Anamnesis' HDFS API implementation.



(b) State transitions between uninflated and inflated blocks based on data requests and memory eviction respectively.

Figure 5.4: Memory-based data block storage using sketched data.

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 5.1).

Figure 5.4a 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 5.4b, 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 5.5 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.

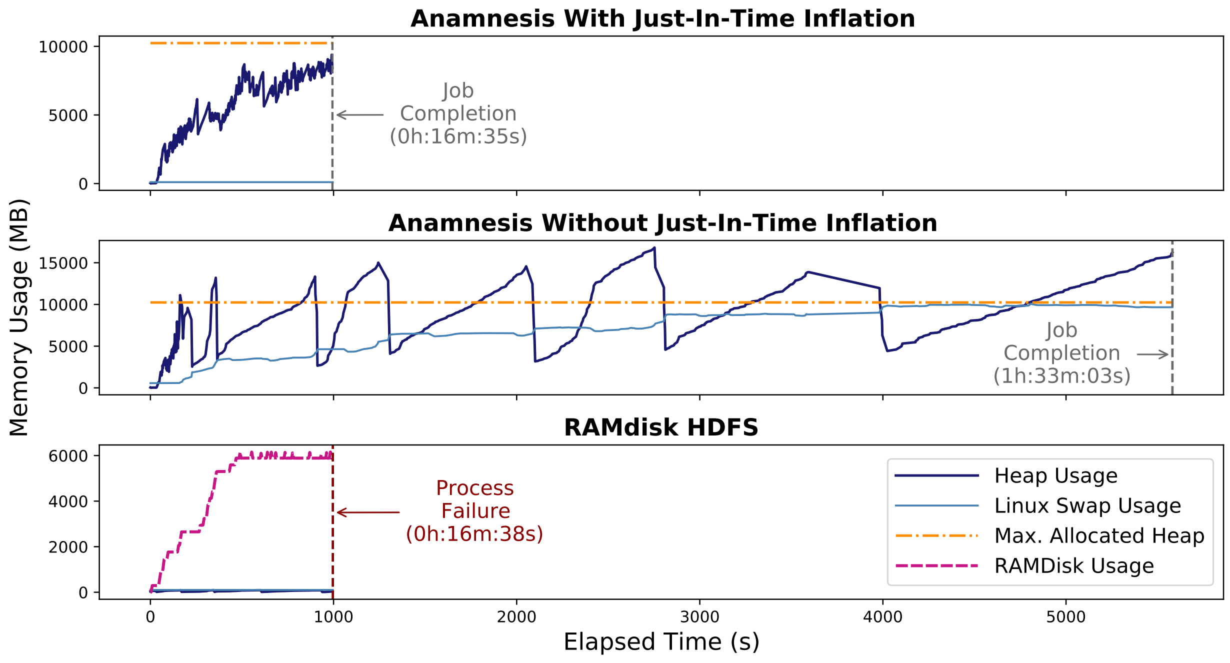


Figure 5.5: Evaluation of memory contention techniques comparing Anamnesis with RAMDisk HDFS.

5.2 Filtering Spatiotemporal Data Blocks

To provide efficient spatiotemporal analytics over data blocks Atlas maintains a collection of efficient indexing data structures. Our methodology includes support for processing index updates efficiently, and enables for robust, efficient query operations. This is important because inefficiencies in index construction may halt subsequent analytics. On overview of these indexing data structures is provided in Figure 5.6.

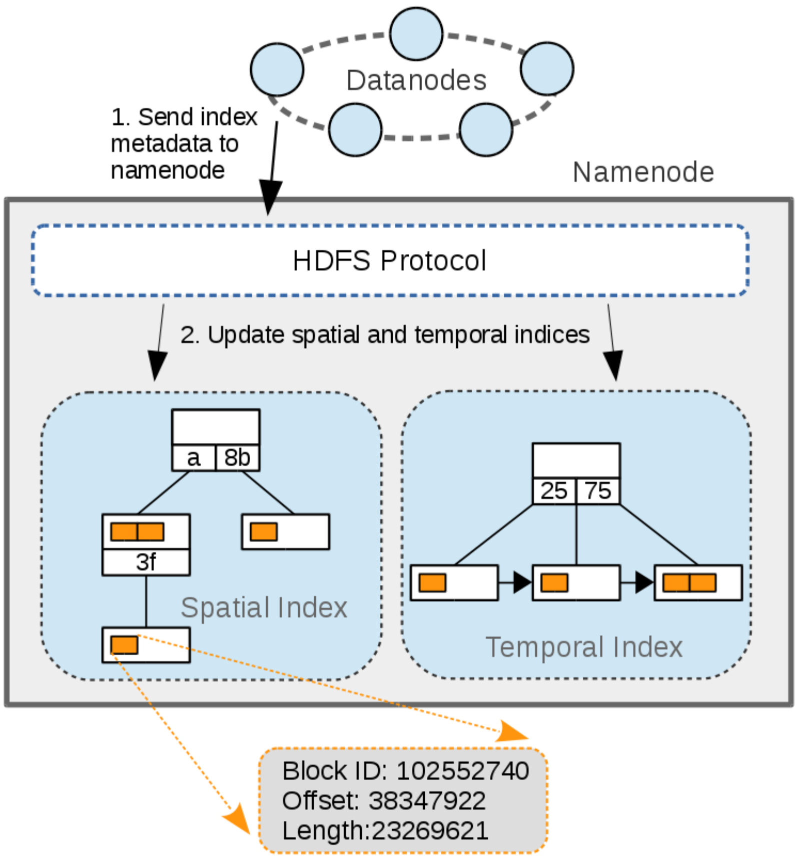


Figure 5.6: Atlas' namenode architecture and maintenance of spatiotemporal indices.

The Atlas spatial index is a radix tree over block geohashes. Each node in the tree contains a list of block ID, offset, and length tuples representing data belonging to the corresponding nodes geohash. Queries over the spatial index return a superset of requested data to ensure complete coverage. For example, a data query for the geohash

8bce will return data that is indexed as 8b, because the later may contain data bounded by the former. The use of this structure ensures simple identification of dataspace subsets satisfying the geohash predicate.

A temporal index is maintained using a B+-Tree with start and end timestamp ranges for each block. B+-Trees are self-balancing, space-efficient data structures that are optimized for efficient range queries. This allows Atlas to quickly identify blocks that contain data for a specific temporal range.

5.2.1 HDFS Compliance

As part of this study, we have incorporated HDFS compliance into Atlas to effectively serve spatiotemporally indexed data blocks. Atlas incorporates communication protocols, message exchanges, and discovery operations that are required (and supported) by canonical HDFS. This allows seamless interfacing with analytics tools. We depict the position of AtlasFS as substituted for HDFS in the analytics stack in Figure 5.7. To provide this spatiotemporal support within the bounds of HDFS protocol we have implemented three novel extensions; (1) Extension of HDFS' *storage policy* framework, (2) embedding spatiotemporal queries with HDFS URLs, and (3) extending the block ID to encode spatial filters.

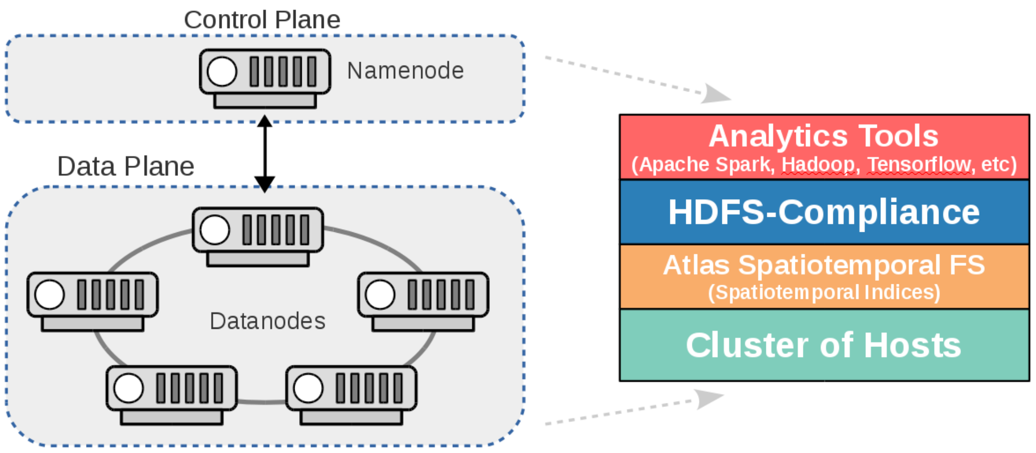


Figure 5.7: The Atlas architecture highlighting the hierarchy of provided protocols and application interfaces.

Canonical HDFS enables each file and directory to be tagged with a *storage policy*. This tag provides system hints as to where data should be available within the memory hierarchy, for example: disk, RAMdisk, RAM, etc. Atlas extends this tag to include the data format for that particular directory. Specifically, it explains where to find that spatial and temporal attributes within each observation. This allows Atlas to support a variety of file formats including CSV, WKT / WKB, NetCDF, and HDF5.

Atlas file URLs may contain an embedded query, similar to the HTTP protocol, which can include spatial (geohash) and / or temporal (numeric) filtering criteria. Queries may be applied to directories, returning blocks from children that satisfy the criteria, or files. An example of a URL with an embedded query is "hdfs://noaa/lattice-126.csv+geohash=8bc&timestan
In this example we are requesting the file "/noaa-imputed/data0/lattice-126.csv" and filtering geohash equal to `8bc` and timestamp greater than "1419897600". Spatial and temporal filtering operands that are supported by Atlas are presented in Table 5.4. Evaluation of queries is performed at the namenode in the "getBlockLocations" RPC call, transparently returning all block IDs (and metadata) that satisfy the query.

Table 5.4: Spatiotemporal filtering operands that may be embedded within Atlas' request URLs.

Filter Type *(Unit)*	**Operand**
Spatial *(Geohash)*	Equality *(=)* Non-Equality *(!=)*
Temporal *(Timestamp)*	Less Than *(<)* Less Than or Equal *(<=)* Greater Than *(>)* Greater Than or Equal *(>=)*

We also extended the HDFS block ID to encode spatial scopes. This is necessary to provide efficient data retrieval, allowing sequential disk reads to process any spatial subset of a block. For example, a query which requests non-consecutive spatial partitions of a stripped set is able to be performed in a single request with our solution. Without it, a request for each spatial partition is necessary, incurring excessive overheads due to disk head movements. HDFS block ID's are 64-bit unique values. In Table 5.5 we have outlined our partitioning of these bits to encode spatial scopes. Our encoding scheme relies on the fact that Atlas can include 16 unique geohashes within each block (hence using a base 16 variant of the geohash algorithm). Figure 5.8 provides an example geohash and the decoded spatial scope.

Table 5.5: The bit sequence index layout of Atlas' HDFS block ID extensions.

Bit Indices	Description	Key	Value
1	Index Flag	0 1	Non-indexed Indexed
2 - 27	Unique ID	-	-
28 - 31	Geohash ID code	0 - 8 9 - 15	Include 0 - 8 Exclude 1 - 7
32 - 64	4 bit Geohash ID's	-	-

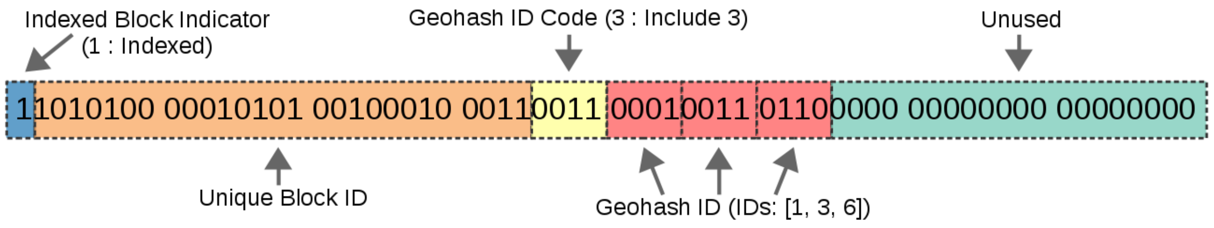


Figure 5.8: A sample HDFS block ID broken down according to the Atlas spatial encoding scheme.

Microbenchmark: In this experiment, we performed spatiotemporal filtering operations over the EPA dataset using Apache Spark. We provide analyses over a variety of spatiotemporal scopes. Spatial filtering is performed using geohashes and the corresponding latitude and longitude boundaries, temporal filtering evaluates over the bounding timestamps, and spatiotemporal operations leverage a combination of the aforementioned criteria. Within Spark, we read the base dataset into a DataFrame, filter based on the necessary criteria, and perform a count operation over the results. Counting the resulting observations is necessary because Spark's lazy evaluation of DataFrames, where operations are only performed when necessary.

In Table 5.6 we present the results of our filtering experiments; reporting durations, disk I/O, and network I/O. We see that Atlas consistently outperforms canonical HDFS. Filtering duration is reduced by up to 7.9x, 2.8x, and 5.2x for spatial, temporal, and spatiotemporal filtering criteria respectively. Both disk I/O and network I/O are **reduced by up to 3 orders of magnitude**. The variance in duration between spatial, temporal, and spatiotemporal filtering is caused by the amount of data that needs to be read from disk. We observed that Atlas temporal filtering provides an order of magnitude better disk I/O performance (for similar filtered data sizes). This is because Atlas' spatial indices provide finer querying granularity over blocks.

Table 5.6: Comparison of Atlas with canonical HDFS for spatiotemoral dataset filtering.

Filter Type *(Unit)*	**Filter**	**Duration**		**Disk I/O**		**Network I/O**	
		Atlas	**HDFS**	**Atlas**	**HDFS**	**Atlas**	**HDFS**
Spatial *(Geohash)*	8e80	9.97s	1:17.3s	126.4MB	193.9GB	320MB	188.9GB
	a76b	10.9s	1:15.3s	130.7MB	194.7GB	316.4MB	188.2GB
	b2296	10.9s	1:26.5s	130.7MB	195.3GB	317.0MB	188.5GB
Temporal *(Duration)*	2 Weeks	28.6s	1:20.7s	15.8GB	195.0GB	17.4GB	186.4GB
	1 Week	28.5s	1:11.9s	15.7GB	195.0GB	17.2GB	188.6GB
	1 Day	29.6s	1:17.4s	17.8GB	194.9GB	19.2GB	188.7GB
Spatiotemporal *(Geohash - Duration)*	8bce - 2 Weeks	15.9ss	1:23.6s	344MB	195.2GB	582.4MB	187.6GB
	9b - 1 Week	18.9s	1:14.2s	568.9MB	193.8GB	819.6MB	187.0GB
	a53 - 1 Day	19.2s	1:21.3s	1.2GB	195.8GB	1.5GB	188.0GB

These improvements can be attributed to (1) spatiotemporal indices within Atlas providing targeted data access and (2) support for sequential disk reads that circumvent the overheads from disk head movements. Furthermore, HDFS-compliance allows Atlas to seamlessly interface with the Apache Spark framework, facilitating adoption into existing workflows.

5.3 Spatiotemporal Image Queries

Analytics over satellite imagery benefit from inclusion/exclusion of certain portions of the dataspace. We support expressive queries aligned with the characteristics of the data to support model construction over the underlying dataspace. This becomes increasingly important when considering the iterative nature of the analytics process which may involve diverse temporal queries for a particular spatiotemporal scope. Challenges in efficiently filtering large datasets are exacerbated in distributed environments where approaches based on distributed indexing and cooperative query evaluation have been studied. Distributed data indices are expensive in terms of both memory and computation. Additionally, solutions where the indices and the underlying data are resident on disparate nodes introduces overheads stemming from indirections and network transfers. Cooperative query evaluation introduces processing inefficiencies with nodes participating in queries even when they do not have relevant data. Therefore, we alleviate inefficiencies by maintaining an in-memory index over dataspace metadata. Additionally, we ensure effective filtering due to search space reductions, distributed evaluations, and fast traversal of indexing data structures.

Any node within our cluster can serve as the conduit for query evaluations, as data is distributed deterministically. We use B+-Trees for efficient range-based queries over data features, including image timestamps, cloud coverage, and pixel coverage. We also employ Radix Trie's for fast (and recursive) geocode prefix matching, meaning we can quickly identify regions starting with a given character string. Finally, we perform query evaluations in parallel on each cluster node; results from the nodes where the queries are evaluated are streamed to ensure that data retrieval is not a bottleneck.

We support queries involving predicates that span the following characteristics.

- **Chronological range based on open or closed bounds:** Filter images by temporal bounds, either end may be open.

- **Spatial range:** Identify all images within the given spatial bounding box. This query may be recursive (i.e. all subregions) or not (i.e. only the queried region).
- **Cloud coverage:** Retrieve imagery where the occlusion due to clouds is above the specified threshold.
- **Pixel coverage:** Source images may not provide data for the spatial extent represented by a geocode. Pixel coverage refers to percentage of image representing valid data.
- **Dataset:** Given that the system supports a variety of data sources, users may specify the satellite imagery of interest: MODIS, NAIP, NLCD, or Sentinel-2.

5.3.1 Complementing Dataspace Features

We also support data wrangling operations that complement our dataspace queries. Specifically, they remedy situations where simple filtering operations are not powerful enough to identify and retrieve complex data subsets. For example, co-processing of multiple image sources where each has different spatiotemporal granularity and pixel resolutions. We logically partition these operations into three subsets, namely bolstering filtering criteria, reconciling image extents, and aligning data from diverse spatiotemporal scopes.

As partitioned images are ingested, we extract and organize metadata to support fast query evaluations. Image attributes are typically drawn from source metadata, which includes spatial and temporal ranges, dataset identifiers, etc. We supplement these attributes with additional metadata relevant for building spatial deep learning models. In particular, we compute two additional attributes as satellite imagery are ingested into the system: pixel coverage and cloud coverage. Pixel coverage is the percentage of pixels within the image (across bands) that contain valid information. Source images may misalign with geocode spatial bounds. As a result, spatially partitioned images may contain partial information with padded 'fill' value pixels. This may be seen in Figure 3.5, which shows four separate NAIP images with less than 100% pixel coverage percentages combined into a full image. Cloud coverage refers to the percentage of pixels that are identified as clouds within an image. Clouds result in occlusions, and their opacity preclude analyses for the areas that they impact. Therefore, for each pixel we use a likelihood estimate to identify the probability of cloud coverage. We make aggregate information available for filtering as a percentage of cloud coverage within an image. This is particularly useful for applications requiring spatial imagery that may be subject to extremes, such as vegetation covers, urban areas, deserts, etc.

Another capability we support is tensor dimensionality reconciliation across bands that comprise the hyperspectral image. As seen in Table 1.2, images from different bands for the same scan may be at different resolutions. In some cases, we may also attempt to perform analyses on data from bands from different satellites. In each case, the tensors (multidimensional arrays) that serve as inputs during processing must be at a desired dimensionality. To solve this issue, we provide image up-sampling and down-sampling functionality which amalgamates data with contrasting resolutions.

Chapter 6

Support for Diverse Analytics

While a vast majority of spatiotemporal analytics can operate using only simple predicate-based filtering, unlocking the full potential of spatiotemporal datasets requires comprehensive analytical operations. These may include computation of distances between spatial objects, computing the length of a line or the area of a polygon, or identifying the intersection between two polygons. Apache Spark is a popular distributed, in-memory analytics suite. It provides robust analytics operations, but is lacking in functionality relating to spatiotemporal data processing. This is becoming increasingly important as spatial data collection increases. In this section we focus on addressing spatiotemporal operation support in the Apache Spark ecosystem.

To support effective spatial data wrangling operations over voluminous datasets within Spark we have implemented AtlasSpark [113], a collection of spatiotemporal extensions to the Apache Spark framework. The position of this framework in the analytics stack is presented in Figure 6.1 It provides three key features: (1) We support a variety of spatial objects. We allow datasets to contain points, lines, polygons, or collections of objects. (2) We include extensive for support computing relationships within or between spatial objects. For example, computing the distance between a point and a polygon, or testing whether a line crosses a polygon. (3) We support datasets encoded in diverse formats. Spatial wrangling operations that we support and designed to be performant and operate over distributed, voluminous datasets. We have extended the JTS library [114] to provide the basis for spatial functionality introduced in this work.

6.1 Data Wrangling Operations

Our AtlasGeometryUDT is an extension of Spark's UserDefinedType abstraction. It enables dataset fields to be parsed into Spark defined geometric objects. We currently

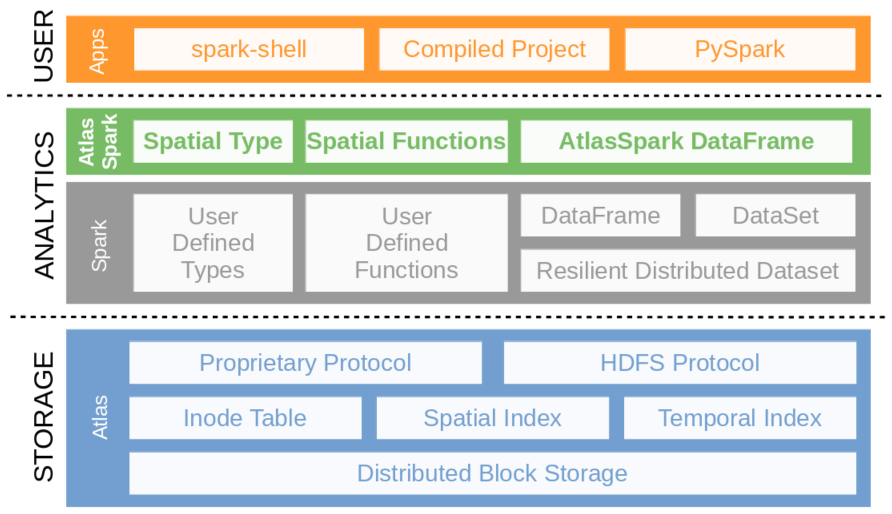


Figure 6.1: AtlasSpark's integration into the analytical hierarchy.

support four different initialization functions, namely BuildPoint, BuildLine, BuildPolygon, and ParseWkt. We have also incorporated support for extensions by simplifying steps needed to process additional data sources.

By extending Sparks UserDefinedFunction API we provide specific spatial functionality over our geometric data type. In Tables 6.1 and 6.2 we outline the diverse collection of spatial unary and binary operations respectively. These extensions may be used during data transformations, filtering, and more. Our suite of data wrangling operations contains numerous return types including booleans, numeric values, and diverse spatial objects.

Table 6.1: Unary operation spatial extensions to Apache Spark.

Data Type	Function Name	Description
Boolean	IsEmpty	Check if the geometry contains data.
	IsSimple	Check if the geometry is simple.
	IsValid	Check if the geometry is topologically valid.
Numeric	Dimension	Returns the dimension of the geometry.
	NumPoints	Returns the number of data points in the geometry.
	Area	Returns the area of the geometry.
	Length	Returns the length of the geometry.
Geometry	Buffer	Computes a buffer area around the geometry with the specified width.
	ConvexHull	Computes the smallest convex polygon which contains all points in the geometry.
	Envelope	Creates a geometry representing the bounding box of the geometry.
	Normalize	Creates a geometry with the normalized form of this geometry.

Table 6.2: Binary operation spatial extensions to Apache Spark.

Data Type	Function Name	Description
Boolean	Contains, Covers, Crosses, Disjoint, Equals, EqualTollerance, Intersects, Overlaps, Touches, Within	Tests whether the geometry satisfies the relationship with the argument geometry.
Numeric	Distance	Returns the distance between the two closest points in the provided geometries.
Geometry	Difference, Intersections, SymDifference, Union	Creates a geometry with the point-set of data as the geometrys relationship.

A Spark Scala example of the aforementioned spatiotemporal functionality is provided in Listing 6.1. When executed; data is read from the Atlas File System, filtered based on the specified polygonal bounds, and the resulting rows are counted. Various components of the code will be referenced in this Section as necessary to clarify and elaborate explanations.

Listing 6.1: A Scala example outlining our spatiotemporal Spark extensions. The listing depicts registration of Atlas components, initialization of a Spark DataFrame over Atlas, filtering within polygonal bounds, and evaluation of the DataFrames observation count.

```
// register atlas spark components
import org.apache.spark.sql.atlas.AtlasRegister
AtlasRegister.init(spark)

// read dataframe from atlas csv file
val df = spark.read.format("atlas")
    .load("hdfs://129.82.208.10/data0/noaa")

// evaluate spark sql statement
df.createOrReplaceTempView("noaa")
var spatialDf = spark.sql("""
␣␣␣␣SELECT␣BuildPoint(_c0,␣_c1)␣as␣point,␣*
␣␣␣␣FROM␣noaa
␣␣␣␣WHERE␣Within(point,
␣␣␣␣␣␣␣␣BuildPolygon((0.0,␣0.0),␣(0.0,␣10.0),
␣␣␣␣␣␣␣␣␣␣␣␣(10.0,␣10.0),␣(10.0,␣0.0)))""")

spatialDf.count
```

Microbenchmark: We have benchmarked our spatial functionality by executing each operation 1 million times over various subsets of our NOAA and EPA datasets. Each experiment has been performed in isolation; detached from the Apache Spark application, but maintaining all Apache Spark field serialization / deserialization operations. The results in a measurement over the operation, excluding as much application overhead as possible.

All reported values are in milliseconds. Tables 6.3 and 6.4 provide information on unary and binary operators respectively. Unsupported operations are represented with a -. Each column represents operations performed on a specific spatial object type(s). The binary information provided in Table 6.4 is labeled as “operand1(operand2)” where the operation is performed in that order. For example, performing the operation “Within” operation under “Point(Poly)” means that a point is within a polygon. The columns for unary operations in Table 6.3 are labeled as a single spatial object. In both tables, empty cells denote operations which do not apply to the corresponding spatial object(s).

We notice variance in operation execution durations. Generally, durations increase as the operation return type moves from boolean, to numeric, to geometric. Additionally, operations performed on polygons tend to be more expensive than lines, with a similar relationship between lines and points. Intuitively, these relationships are reasonable. As testing equality between polygons is comparatively simpler than computing the distance between two polygons, or computing an intersection.

Table 6.3: Performance evaluation of unary spatial operations within Spark.

Operation	Line	Point	Polygon
IsEmpty	847.8	832.8	1123.6
IsSimple	1823.6	803.4	2613.6
IsValid	1232.0	807.2	5125.8
BuildLine	1998.6	-	-
BuildPoint	-	980.8	-
BuildPolygon	-	-	2761.4
Area	-	-	1427.0
Dimension	916.8	851.8	1391.8
Length	917.0	-	-
NumPoints	924.2	880.2	1569.2
Buffer	48318.0	9534.0	10856.0
ConvexHull	1584.4	974.4	1885.0
Envelope	1010.8	872.0	1367.8
Normalize	946.4	863.2	1448.8

Table 6.4: Performance evaluation of binary spatial operations with Spark.

Operation	Line			Point			Poly		
	Line	Point	Poly	Line	Point	Poly	Line	Point	Poly
Contains	1891.6	1548.8	-	-	1348.8	-	1835.4	1688.2	1876.4
Convers	1680.4	1525.0	-	-	1388.2	-	1886.8	1668.4	1935.0
Crosses	-	-	1727.0	-	-	-	1838.0	-	-
Disjoint	2300.2	1497.4	1809.0	1607.2	1412.2	1751.4	1938.4	1744.6	2003.2
Equals	1589.4	1460.2	1635.4	1508.0	1365.0	1571.2	1764.4	1598.8	1818.8
EqualsTollerance	1585.4	1465.2	1687.2	1631.8	1395.8	1661.6	1894.8	1691.4	1945.2
Intersects	3165.0	1530.0	1797.2	1640.4	1428.0	1697.2	1924.8	1728.0	2028.0
Overlaps	3090.0	-	-	-	-	-	-	-	2318.4
Touches	3664.4	1546.6	1793.6	1622.0	1426.6	1712.6	2044.4	1690.4	2042.4
Within	1848.2	-	-	-	-	1736.2	-	-	2019.0
Distance	2069.8	1728.4	2451.4	1825.8	1523.2	2016.2	2748.0	2015.4	2950.2
Difference	10857.0	6651.2	10539.6	7618.0	2759.0	6542.8	12451.8	7027.8	11717.6
Intersection	11647.4	6304.6	10336.6	5848.6	2793.8	6621.0	9098.4	6559.2	10809.8
SymDifference	12228.4	6765.2	11808.6	6218.0	3002.2	7322.6	10805.6	7336.8	11763.0
Union	13795.0	7317.8	12123.0	9347.2	3100.2	7444.4	13011.4	7259.8	11151.6

6.2 Scheduling with Data Locality

The Spark framework relies on data partitioning algorithms to effectively distribute and orchestrate analytics workloads. These algorithms split source datasets into many smaller chunks that are then distributed. Data partitioning and distribution allows Spark to process the data in parallel. Data partitioning does not have a one-size-fits-all solution, because each scheme has its own inherent advantages and disadvantages for various analytics tasks – no scheme performs well in all scenarios.

Spark's default partitioning scheme for HDFS data is not suited for spatiotemporal data. By default, Spark partitions HDFS data along block boundaries. However, this scheme is inefficient for spatiotemporal access patterns that are triggered during data wrangling and analytics. The default scheme entails large amounts of data transfer between nodes for many operations (i.e. wide transformations).

We have extended Spark's DataSourceV2 framework to partition datasets aligned with the spatiotemporal storage policies provided by Atlas. The DataSourceV2 implementation presents many abstraction improvements, including support for reads / writes with data streams and executing data filtering operations at the data source. Atlas storage policies are designed to address the competing pulls of data locality and dispersion. In Atlas, data within a particular spatiotemporal space is dispersed over a small subset of machines, facilitating parallel processing while minimizing data movements. By aligning our partitions with Atlas we ensure (1) there is little data movement during analytics, because processing is scheduled on data resident nodes and (2) many analytic operations may be performed with data locality; favoring narrow, rather than wide, Spark transformations during evaluation.

Micro-Benchmark: In this experiment we deployed Atlas and Spark on a 50 node cluster, where one machine houses the Atlas namenode and Spark master and the other 49 are Atlas datanodes and Spark workers. We dispersed over 1TB of data from our NOAA dataset over this cluster to profile its distribution effectiveness. Figure 6.2 plots the combined size of data partitions scheduled at each of the Spark worker hosts during analytics over the entire dataset. We see that even though Atlas preserves spatiotemporal proximity during data distribution, the data partitions are not skewed, but are rather evenly distributed over the cluster. This results in load-balanced processing, reducing hot-spots during analytics.

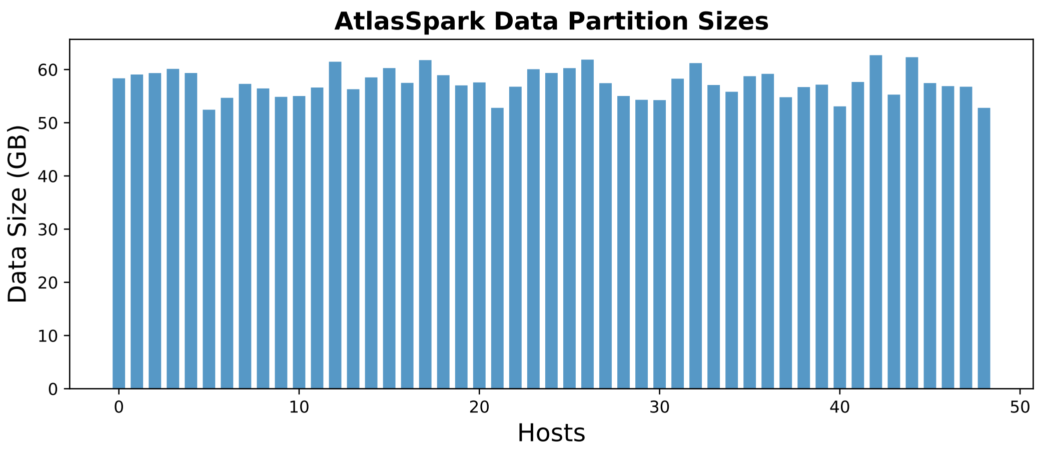


Figure 6.2: Data partitions for Atlas analytics using our spatial extensions in Spark.

6.3 Incremental Operation Optimizations

To reduce duplicate operations, the Apache Spark framework relies on lazily-evaluated operations. Meaning instead of evaluting operations in real-time Spark creates and consults an abstract syntax tree (AST) of operations. These operations may include field projections, dataset filtering, and data reads from a variety of sources. Spark dynamically performs a sequence of AST optimizations before evaluation to facilitate more efficient analytics. Catalyst is the AST optimizer distributed with Spark. It maintains a registry of predefined rules, which may be applied iteratively when evaluating the AST. Catalyst rules include boolean simplification, constant folding, column pruning, operation propagation over data joins and field aggregations, and many more.

A key Catalyst optimization that we target is predicate pushdowns. The predicate pushdown rule attempts to push dataset filtering operations to data sources by propagating filters down to data sources in the AST. The pushdown feature provides two objective advantages. (1) Data source implementations are often optimized to perform filtering by employing various data indexing structures and techniques. In some cases, this entails Spark iterating over each row and evaluating the filter individually. (2) It reduces data movements and network I/O. By performing filtering earlier in the AST evaluation, unnecessary data (i.e., data that does not satisfy the filter predicated) movement between stages is reduced.

We addressed a number of challenges to incorporate support for spatiotemporal predicate pushdowns within the Catalyst Optimizer. Currently, Spark lacks support for "push-down" of UserDefinedFunctions. Furthermore, the diversity of UserDefinedFunctions introduces additional complexities when converting to filters that are aligned with Atlas.

To enable spatiotemporal query definitions using our UserDefinedFunctions we register a series of Catalyst optimization rules. These rules inject filters into the AST allowing them to be propagated using Catalyst Optimizer's predicate pushdown rules. For example, in Listing 6.1 lines 11-16 we use a Spark SQL query for filtering the NOAA dataset on data "Within" the defined polygon, with vertices identified by `[latitude, longitude]` pairs of `[0,0]`, `[0,10]`, `[10, 0]`, and `[10, 10]`. We created and registered a Catalyst optimization rule to identify the "Within" functions where the Atlas spatial index field(s) are bounded by another geometry, in this case the aforementioned coordinates. We then compute the smallest bounding geohash for these coordinates, for example dac, and inject a filter into the AST for data within that bound, namely "atlasGeohash = dac". Temporal filtering and bracketing is evaluated similarity. This new equality filter satisfies the requirements for Catalyst's predicate pushdown rules, and can therefore be propagated to the Atlas data source and evaluated accordingly. A list containing the bounding polygon defining the geohash for each of our supported spatial operations is provided below.

1. **Contains:** The bounding polygon by which "Contains" is processed.

2. **Covers:** The polygon which "Covers" other spatial objects.

3. **Distance LessThan / LessThanOrEqual:** A buffer computed with the provided distance from the source spatial object.

4. **Equals / EqualsTollerance:** An outer bounding polygon for which equality is tested.

5. **Within:** The polygon which other spatial objects are tested "Within".

Additionally, we have implemented two classes of filter combination rules over our spatiotemporal bounds. The first is the aggregation of spatial or temporal bounds. For example, consider a data source that requires filtering for "atlasTimestamp >1564701343" and "atlasTimestamp >1064701343". Instead of executing two separate queries, we aggregate the filter to the less restrictive filter, namely "atlasTimestamp >1064701343". The second is pruning AST branches when filter aggregations will result in no data. For example, combining the two filters "atlasGeohash = 8bce" and "atlasGeohash = a97". There are no valid data that satisfy both filters, therefore evaluation is unnecessary.

6.4 Comparison with Canonical Spark and HDFS

6.4.1 Histogram Generation

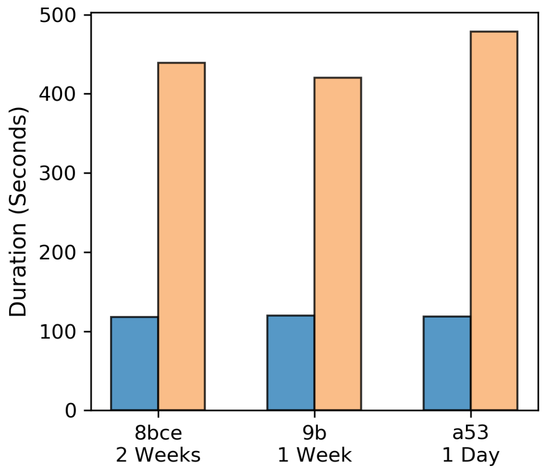
To highlight the effectiveness of spatiotemporal analytics using Atlas we have chosen to construct a histogram over a single feature within a spatiotemporal subspace. The histogram algorithm is a common analytics operation to estimate the distribution of feature values. We experimented using the NOAA dataset, specifically targeting the surface temperature feature and evaluating under a variety of spatiotemporal scopes, presenting geographically diverse regions with varying durations.

In Figure 6.3 we present our results comparing Atlas with canonical HDFS. Specifically Figures 6.3a, 6.3b, and 7.1b provide the analytics duration, disk I/O, and network I/O incurred during evaluation respectively. We see that **Atlas facilitates a 4x reduction in duration when compared with canonical HDFS**. Additionally, disk I/O and network I/O are continually reduced by 3 orders of magnitude, making results measurable in single GBs instead of TBs.

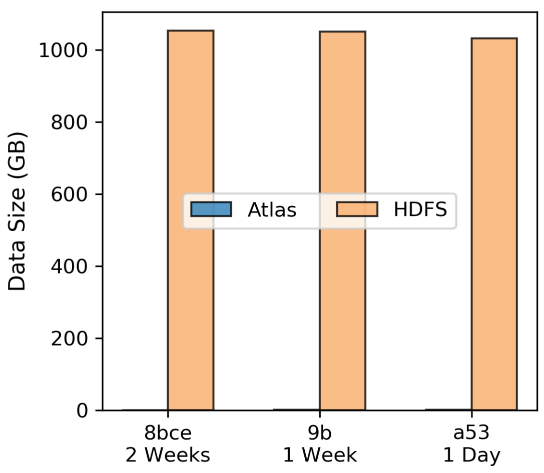
These reductions are directly attributable to Atlas' spatiotemporal optimizations in identifying spatiotemporal subspaces and providing efficient data access patterns. Additionally, the analytics task benefits from Atlas' data distribution policies, facilitating parallel processing enabled by the effective dispersion of data throughout the cluster.

6.4.2 Complex Dataset Filtering

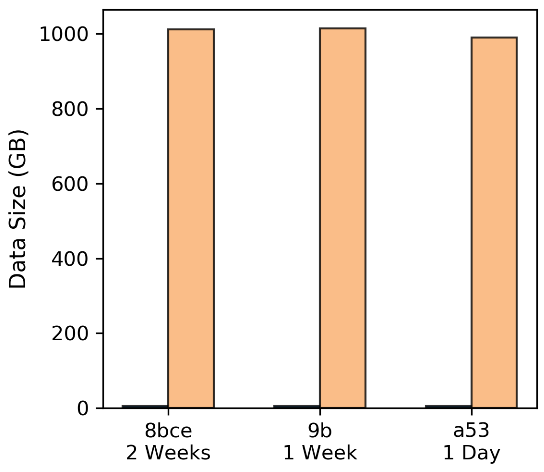
We have contrasted the performance of AtlasSpark, our novel Spark spatial extension, with two popular spatial Spark frameworks. GeoSpark [71] and Magellan [115] facilitate



(a) Atlas reduces analytics duration by up to 4x compared to canonical HDFS.



(b) Disk I/O is consistently reduced by 3 orders of magnitude.



(c) Network I/O is measurable in single GBs instead of TBs after a 3 order of magnitude reduction.

Figure 6.3: Duration, disk I/O, and network I/O incurred during histogram evaluation over a single dataset feature comparing Atlas with HDFS.

efficient spatial retrievals by dynamically computing spatial indices over RDD's using R-Trees and Z-Order curves respectively.

In this experiment we performed spatially constrained range queries over the NOAA dataset. Since the NOAA data comprises U.S.-based vantage points, we restricted our spatial queries within a latitude/longitude envelope governing the continental US. Query predicates were formulated using geohash precisions of length four, giving each query a +/- 0.087 and +/- 0.18 latitude and longitude error respectively. For each platform evaluation we performed 200 randomly defined spatial queries. Since the aforementioned continental US envelope contains 1722 unique geohashes, each platform evaluation provides 12% coverage of the dataspaces spatial scope.

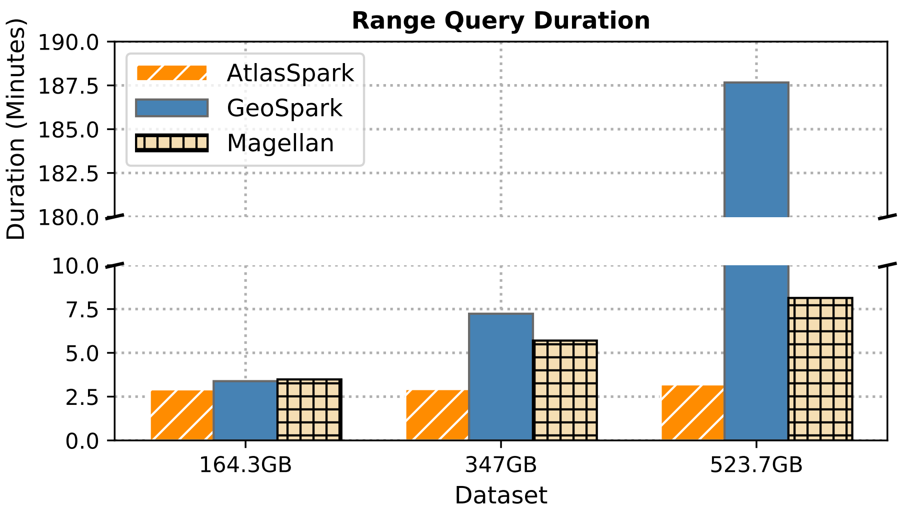


Figure 6.4: Duration of spatially bounded range queries for AtlasSpark.

Spatial range queries are a popular metric for evaluation and comparison of spatial analytics frameworks. Current surveys, such as [116], effectively employ them to contrast framework performance statistics. We evaluated spatial range queries under three distinct temporal ranges of the NOAA dataset; namely 1 month, 2 months, and 3 months – these queries effectively represent data subsets of sizes 164.3 GB, 347 GB, and 523.7 GB respectively. These larger dataset sizes seek to profile AtlasSpark's ability to maintain performant analytics for datasets that are too large to fit into memory.

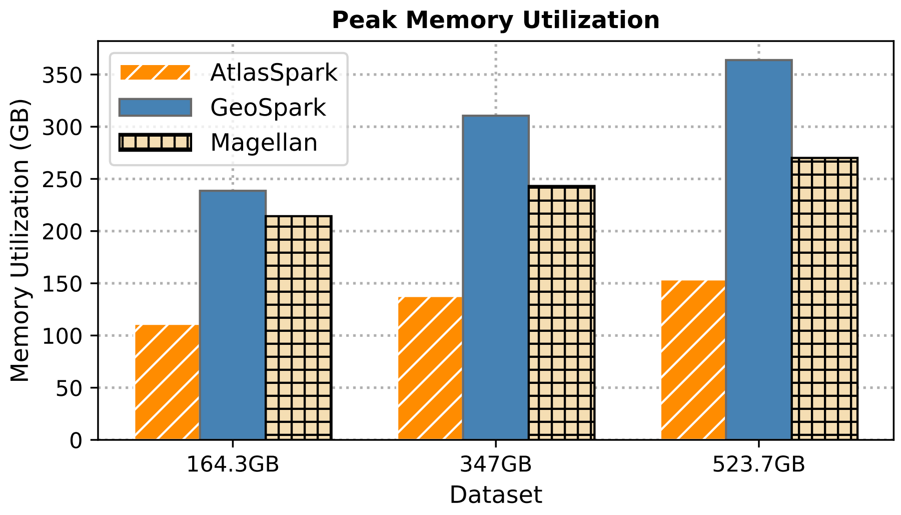
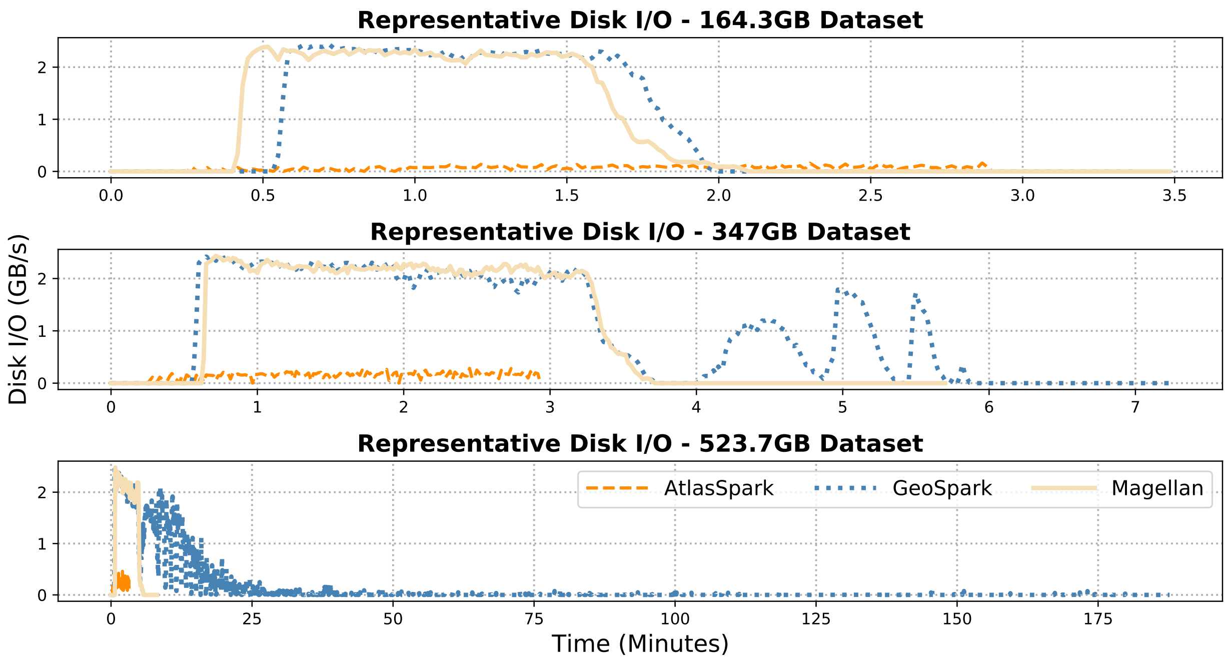


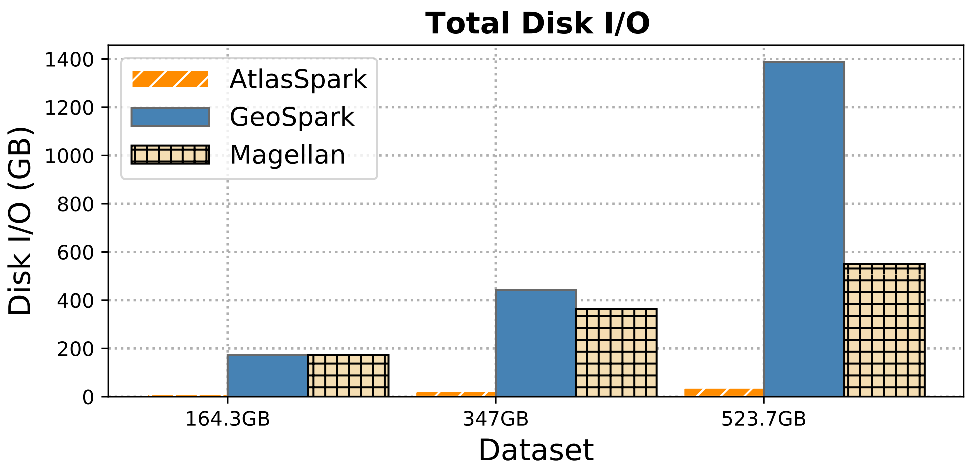
Figure 6.5: Evaluation of cluster memory utilization during range queries in AtlasSpark.

In Figure 6.4 we display range query duration for the three platforms for each NOAA temporal range dataset. We see that AtlasSpark consistently outperforms both GeoSpark and Magellan. In particular, these benchmarks show that **Atlas outperforms Geospark by a factor 1.2x, 2.6x, and 62.1x; AltasSpark also outperforms Magellan by a factor 1.2x, 2x, and 2.7x for these datasets**. These performance improvements are attributable to AtlasSpark's targeted data retrieval, where the system leverages the underlying spatiotemporal indices to perform efficient data retrievals unlike GeoSpark and Magellan which rely on dynamically indexing underlying datasets for queries.

We also profiled peak memory utilization metrics for each experiment; these are depicted in Figure 6.5. As dataset sizes increase, peak memory utilization for the frameworks increases as well. At the largest dataset the GeoSpark framework *breaches* the cluster's available memory threshold at 300GB. The corresponding analytics durations increases steeply in GeoSpark as a result (see Figure 6.4) since expensive operations need to be performed to determine data memory residency. Furthermore, AtlasSpark consis-



(a) Representative disk I/O during analyses.

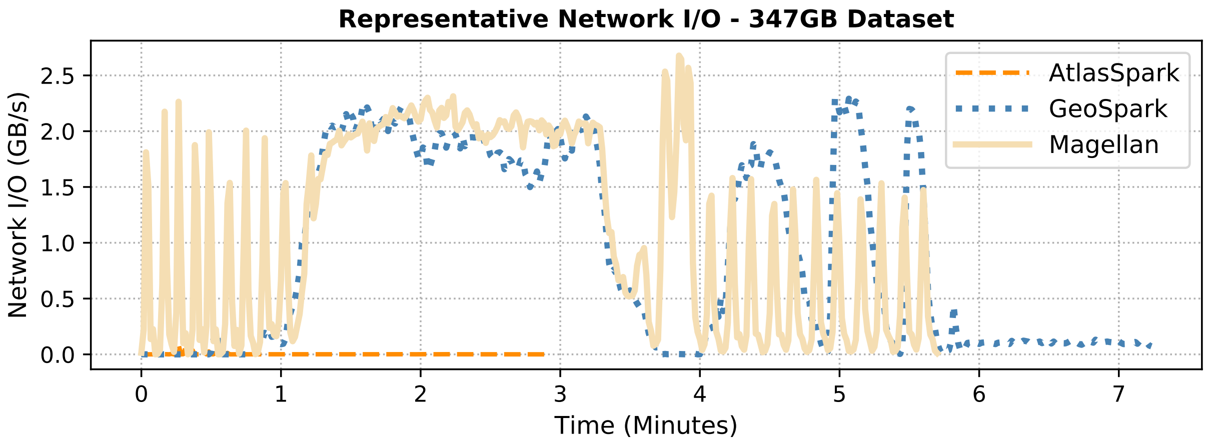


(b) Total disk I/O incured over cluster nodes during analytics.

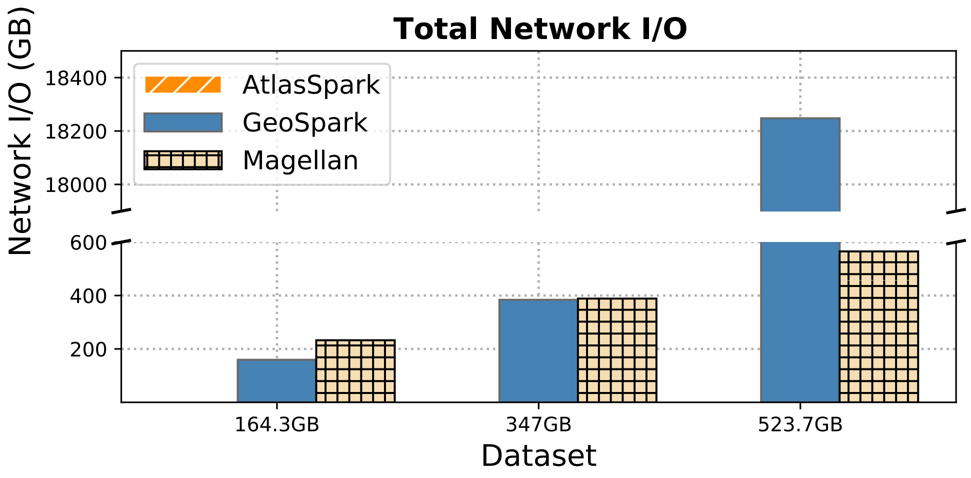
Figure 6.6: Disk I/O for spatially bounded queries comparing AtlasSpark, GeoSpark, and Magellan.

tently maintains a lower memory footprint – **up to a 57.5% and 48.1% reduction in peak memory utilization compared to GeoSpark and Magellan respectively**.

In Figures 6.6a and 6.6b we depict both the representative, and total, disk I/O incurred during analytics respectively. **AtlasSpark reduces disk I/O by a factor of 18x over GeoSpark and by a factor of 14x over Magellan respectively**. Computation of indexing structures in GeoSpark and Magellan entail reading the entire dataset. This spike in disk I/O is easily seen in the beginnings of evaluation in each tier for the representative graphs. Alternatively, AtlasSpark requires significantly less disk I/O by leveraging the underlying spatiotemporal indices. It is also important to note that both representative and total network I/O for each experiment is similar to the aforementioned disk I/O metrics.



(a) Representative network I/O for the 347GB dataset.



(b) Total network I/O for 3 different datasets.

Figure 6.7: Network I/O incurred during spatial range queries using AtlasSpark, Magellan, and GeoSpark.

Finally, we present representative and total network I/O in Figures 6.7a and 6.7b respectively. Figure 6.7b presents the total cluster network I/O for the experiment with each dataset, whereas Figure 6.7a only provides a representative look at a single dataset. The representative network I/O data shows strong similarity with representative disk I/O for the provided 347GB dataset in Figure 6.6a, this characteristic carries for all datasets. We see that **Atlas Spark incurs just 500MB of total network I/O, reducing network I/O by 4 and 5 orders of magnitude when compared with Magellan and Geospark respectively**. Again, this reduction is a construct of AtlasSpark's ability to leverage existing spatial indices instead of dynamically constructing the necessary data structures.

Chapter 7

Effective Model Training

Spatiotemporal datasets offer new and interesting avenues for predictive analytics, where we see the relative increases in data volumes and diversity driving many research efforts. In these situations, model accuracy typically correlates with the volume of data processed, meaning insights become more precise as training datasets scale. This introduces a variety of challenges in the distributed orchestration and synchronization of training efforts.

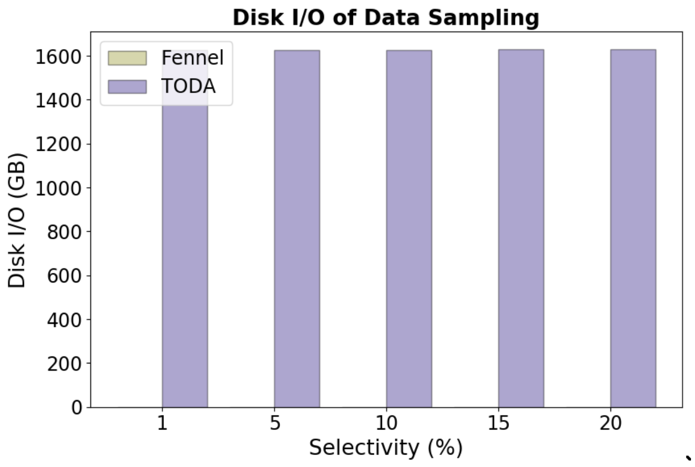
In this section we explore techniques for effective model training over spatiotemporal datasets. This includes processing feature-based and hyperspectral imagery datasets. To this end, we explore:

1. Effectively sampling datasets from sketched data, mitigating inefficient disk access while enabling variable size datasets.

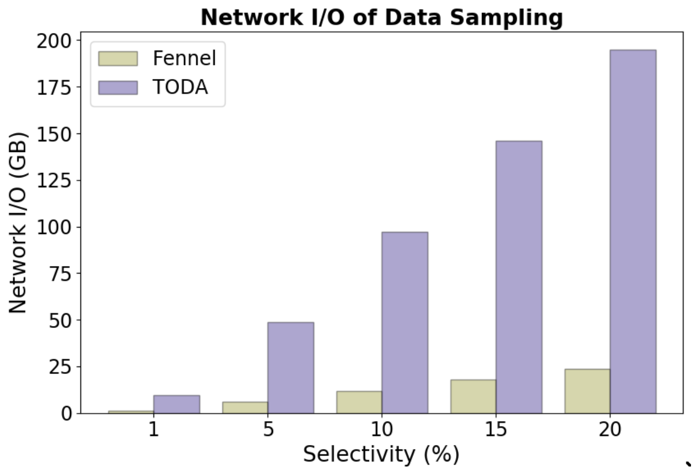
2. Training an ensemble of models where each specializes on a specific spatial extent, relying on data locality to mitigate data movements during the training stage.

7.1 Training with Sketched Data

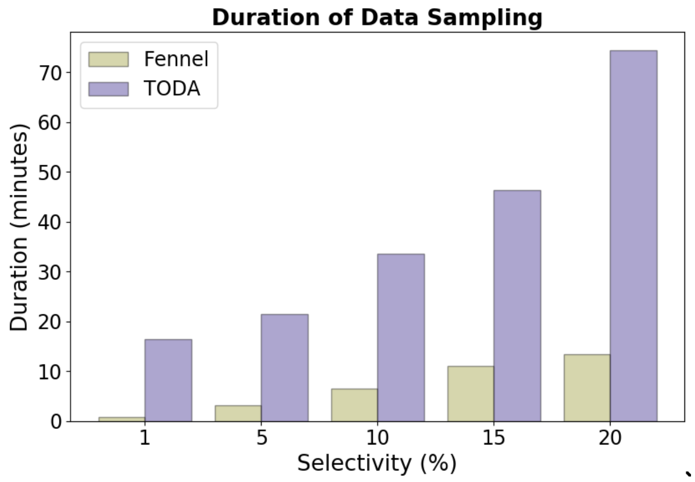
Model training is computationally expensive, dataset sizes and distributed analytics environments introduce problems of scale. Sketched data offers numerous advantages over traditional observational stores. (1) Operation over in-memory datasets mitigates disk access, which are several orders of magnitude slower. (2) The Fennel sketch (i.e., DFVs) store summary statistics of the feature space. Therefore, we may fine tune sample dataset sizes, generating accuracte synthetic datasets which precisely maintain feature distributions and relationships.



(a) Disk I/O incurred during data sampling.



(b) Network I/O incurred during data sampling.



(c) Data sampling duration.

Figure 7.1: Data sampling metrics comparing sketched data with on-disk.

7.1.1 Data Sampling

In Figure 7.1 we explore resource utilization and duration of Fennel data sampling. We compare these results to traditional on-disk analytics (TODA). Model training times are identical between the approaches for each selectivity interval so we have focused on reporting the variances in data sampling. For these experiments we monitored disk I/O, network I/O, and duration based on various sample sizes of the NOAA dataset.

Figure 7.1a reports disk I/O during data sampling; as can be seen Fennel requires no disk I/O. Figure 7.1b reports on network I/O between the Storage and Analytics Clusters. We see that Fennel significantly reduces network I/O. **For each selectivity test Fennel shows an 88% reduction in network I/O.**

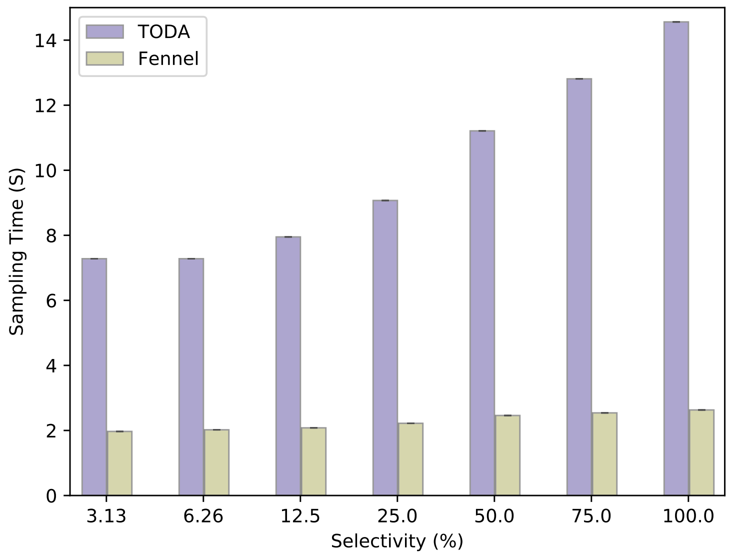
Figure 7.1c displays the overall duration for data sampling. **We observe a 76% - 92% reduction in data sampling times.** However, the trade-off between savings of disk / network I/O and time required to generate accurate synthetic datasets is tunable for higher performance.

Figures 7.2a and 7.2b present sampling times for varying selectivity over the Million Songs and Texas datasets respectively. We see similar improvements as in Figure 7.1c showing that Fennel's advantage is not unique to the NOAA dataset.

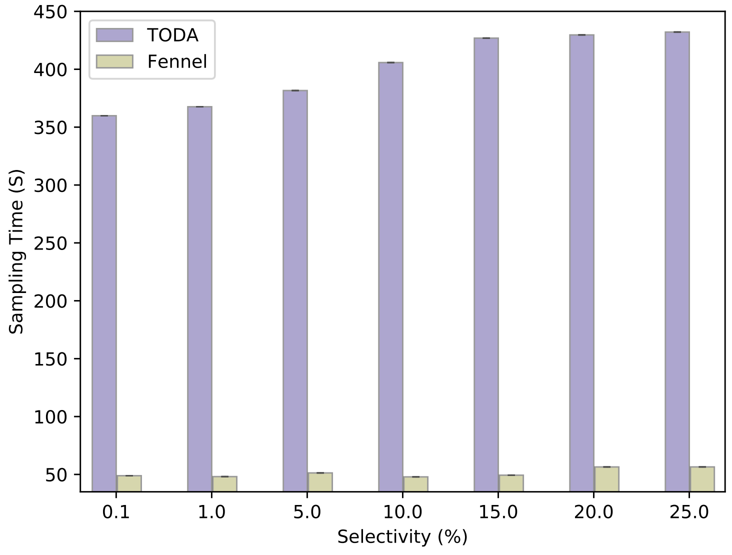
7.1.2 Model Accuracy

We conducted experiments to assess how much accuracy is sacrificed by using the Fennel sketch that significantly reduces training times. We trained models on data sampled from the actual and the synthetic data generated from sketches and evaluated their accuracy on test data sampled from the actual full-resolution data.

Figures 7.3a, 7.3b and 7.3c contrast the accuracy of models built on actual data with the ones trained on synthetic data for the three datasets. In Figure 7.3a we can see that the models trained on actual and synthetic data of Million Song dataset have similar accuracy for all sampling sizes. In this case, we significantly reduced the training time without



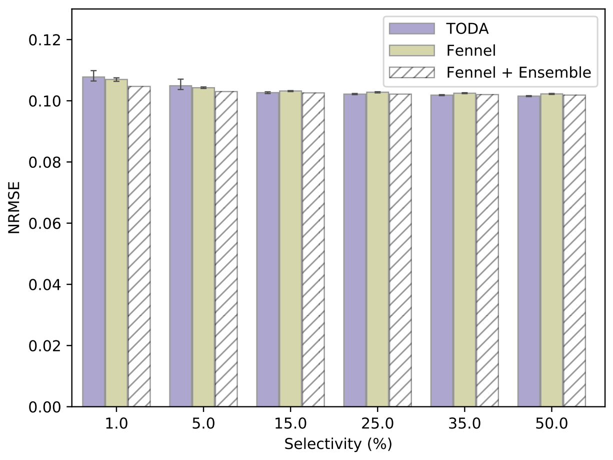
(a) Sampling times for the Million Song dataset.



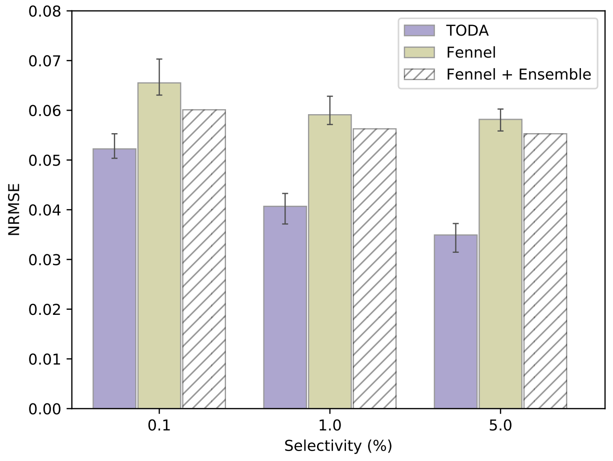
(b) Sampling times for the Texas dataset.

Figure 7.2: Comparisson of data sampling durations for sketched and on-disk data.

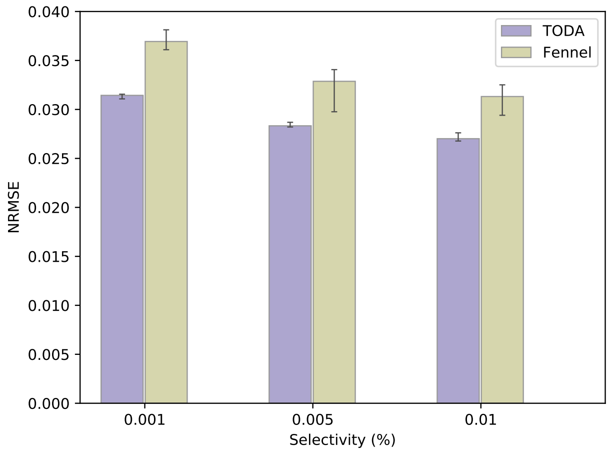
accuracy reduction. For other datasets, models built on synthetic data are less accurate, but as can be seen the differences are not significant. Furthermore, the accuracy improves as the training data sizes become larger.



(a) Million Song Data (*Target: Release year*)



(b) Texas Data (*Target: Disease duration measured in days*)



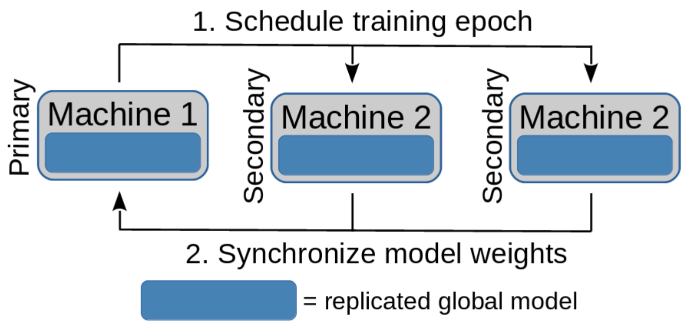
(c) NOAA Data (*Target: Precipitable water measured in millimetres*).

Figure 7.3: Accuracy of models trained on samples from actual and sketched data.

7.2 Orchestration of Deep Learning Workloads

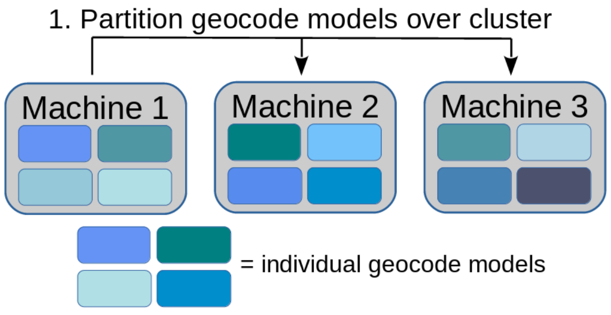
Effective workload orchestration and concurrent execution are necessary to harness resources within a commodity cluster. Orchestration of distributed spatial deep learning tasks introduces a number of challenges. (1) Deep learning operations depend on every layer of the resource hierarchy (i.e., CPU, GPU, RAM, disk, and network) and varying workflow stages have different resource requirements. (2) The model building process is iterative, involving synchronization barriers as model weights are aggregated after each epoch. Given the speed differential across the memory and I/O hierarchy, network I/O costs start to dominate completion times. (3) Right-sizing resource allocations is difficult. Under-provisioning leaves system resources unused, while over-provisioning may further prolong completion times due to excessive disk head movements, CPU context switches, memory pressure, and I/O interference.

Our orchestration (see Figures 7.4b and 7.4c) based on a decentralized DHT leverages traits aligned with the characteristics of the spatial workloads [117]. Specifically, we train a single model for each individual geocode within the system, producing an ensemble of smaller models rather than a large, all-encompassing model. This contrasts with the conventional distributed deep learning architecture (depicted in Figure 7.4a) where a single large model is trained over all data. The conventional approach requires training to be partitioned into iterative epoch scheduling and weight synchronization, which, in unbalanced data distributions, may introduce significant bottlenecks in effective training. Additionally, it allows us to reconcile spatial heterogeneity by having smaller regions tune themselves to particular spatial extents; our performance benchmarks show performance improvements as well. It also permits us to use the DHT's balanced distribution of geocodes as a guide for orchestrating workloads. In addition to balanced evaluation, it ensures training jobs are scheduled with data locality, avoiding unnecessary network transfers and coordination-related overheads.



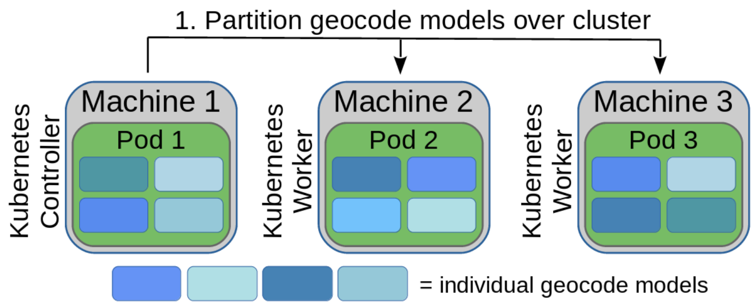
1. Schedule training epoch
Primary
Machine 1
Secondary
Machine 2
Secondary
Machine 2
2. Synchronize model weights
= replicated global model

(a) Distributed Architecture - typical master model.



1. Partition geocode models over cluster
Machine 1
Machine 2
Machine 3
= individual geocode models

(b) Partitioned Architecture.



(c) Containerized Architecture.

Figure 7.4: Architecture of distributed workload orchestration techniques.

We also demonstrate that our workload orchestration scheme is amenable to containerized environments. We containerize the process of training an individual model using Docker [118] and schedule the execution of model training using Kubernetes [119]. We show that Kubernetes is able to take advantage of our partitioning scheme to train multiple data-local models on each machine simultaneously, further optimizing resource use.

We profile the suitability of our methodology for diverse deep learning networks. The networks were chosen based on the types of spatial analyses that are typically performed, and also the representativeness of the structural elements such as depth of the network, layers, regularization schemes, and the types of network. We profile our methodology's impact on training time, resource consumption, and statistical error of models with varying sizes and requirements. These real-world use cases are: (1) Calculation of metrics across bands in hyperspectral imagery (NDVI Prediction), (2) Image translations with preservation of structural characteristics such as roads, buildings, etc. using skip connections (NDMI Generation). (3) Cloud removal from satellite imagery - a relatively new image-to-image translation using spatial attention in a generative adversarial network (SpAGAN).

7.2.1 Duration

We present the duration of model training in Table 7.1. Across models, our partitioned algorithm is faster than the traditional distributed approach. **Explicitly, there is a 6.9x, 1.25x, and 13.3x reduction in training durations for the NDVI prediction, NDMI generation, and SpAGAN models respectively**. The difference may be attributed to the large overhead required in distributing the training and syncing the model across all nodes in the cluster. We also note that **containerizing the partitioned training further reduces training times by 29.8% over a non-containerized partitioned workload and almost 10x over the distributed approach**. This is because Kubernetes leverages composite cluster resource utilization to more effectively schedule training tasks.

Table 7.1: Duration to train models comparing workload orchestration architectures.

Model	Distributed	Partitioned	Containerized
NDVI Prediction	6.8 hrs	59.4 mins	41.7 mins
NDMI Generation	2.0 hrs	1.6 hrs	-
SpAGAN	17.35hrs	1.3 hrs	-

7.2.2 Resource Utilization

We profile CPU utilization in Figure 7.5. The partitioned architecture consistently uses less CPU than distributed. Alternatively, containerizing the environment required considerably more CPU because it is reading datasets for multiple models simultaneously. For each model, we see a strong cyclic relationship on CPU utilization for the distributed architecture. This is due to periods of centralized model weight synchronization.

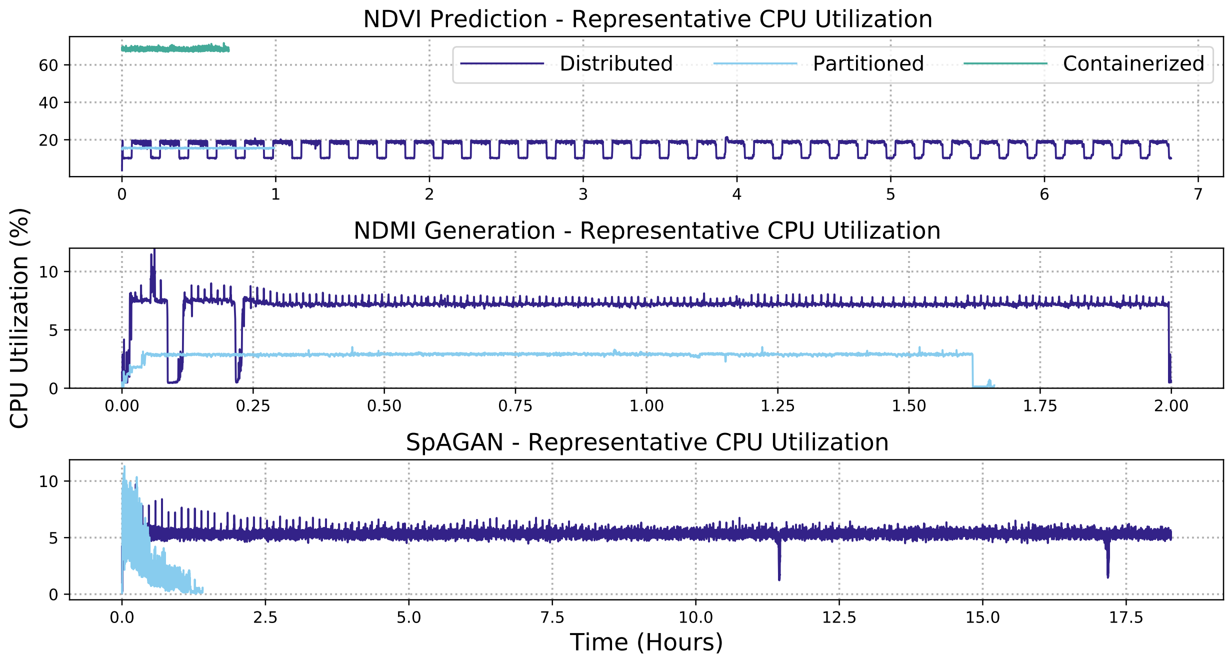


Figure 7.5: Representative CPU utilization during model training with various orchestration schemes.

Figure 7.6 compares GPU use while training. There is a large variance in utilization across model types. Generally this is correlated with the complexity of the model, where complexity increases from NDVI prediction to NDMI generation to SpAGAN. Within models, containerization makes the best use of the GPU by training multiple model simultaneously. Overall, the distributed architecture requires more use than our partitioned approach. This is because the distributed approach exhibits the same periodicity as CPU utilization. Again, this is a construct to model weight synchronization, but results in less efficient use.

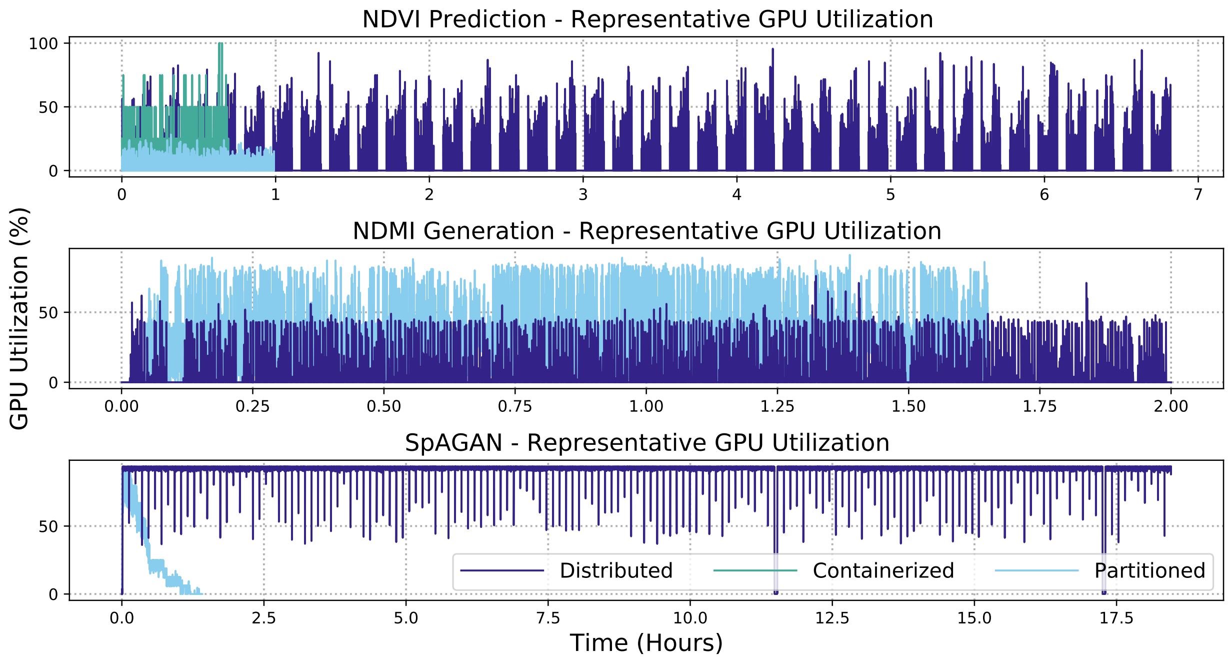


Figure 7.6: Representative GPU utilization during model training with various orchestration schemes.

Total and average network I/O are presented in Table 7.2. Total cluster values may be extrapolated by the number of hosts. Maintaining lower network I/O is important because as clusters scale out, data transfers may become a performance bottleneck. Fortunately, our **partitioned training is significantly lower than distributed training for every model; consistently resulting in a 5 order of magnitude reduction in total network I/O**. Alternatively, containerized training requires each machine to communicate with the Kubernetes Controller, generating a modest amount of traffic. This decrease is established by the collocation of training tasks and data. Our partitioned approach does not require any data movements beyond initializing training tasks. In comparison, distributed training suffers from centralized management and synchronization of model weights.

Table 7.2: Network I/O incurred during model training over spatially partitioned images.

Model	Average Network I/O			Total Network I/O		
	Distributed	Partitioned	Containerized	Distributed	Partitioned	Containerized
NDVI Prediction	460.1KB/s	1.7KB/s	5.6KB/s	113.0GB	6.1MB	14.0MB
NDMI Generation	102.4MB/s	4KB/s	-	737.0GB	23.8MB	-
SpAGAN	100.9MB/s	87.5KB/s	-	6.48TB	430MB	-

7.2.3 Model Accuracy

The accuracy of models is presented in Table 7.3. Figure 7.7 also presents sample result images of the NDMI prediction and SpAGAN models. These visualize image characteristics that are difficult to quantify, for example using our partitioned approach with the SpAGAN model is able to more accurately capture details of the surrounding landscape, like rivers and mountain ridges.

Table 7.3: Model accuracy comparing distributed and partitioned orchestration schemes.

Model	Loss	Distributed	Partitioned
NDVI Prediction	MSE	0.121	0.083
NDMI Generation	MSE	0.00134	0.000253
	MS-SSIM	0.81	0.93
SpAGAN	PSNR	30.93409	31.559914
	SSIM	0.90471	0.92147

Our partitioned architecture performed better, on average, than the distributed approach on the NDVI prediction model. Comparatively, training with the distributed architecture results in a moderately higher loss for the single model, caused by its inability to specialize for certain regions. It should be noted that containerized training is functionally identical to partitioned training, it is merely scheduled differently. Therefore, accuracy measures are directly transferable.

NDMI prediction accuracy is reported using both the mean squared error (MSE) and MS-SSIM metrics. Using MSE, we can measure how much the predicted pixel value differs from the target pixel value, reflecting the water content of that spatial region. To compare the quality of the generated map, we have used multi-scale SSIM metrics that

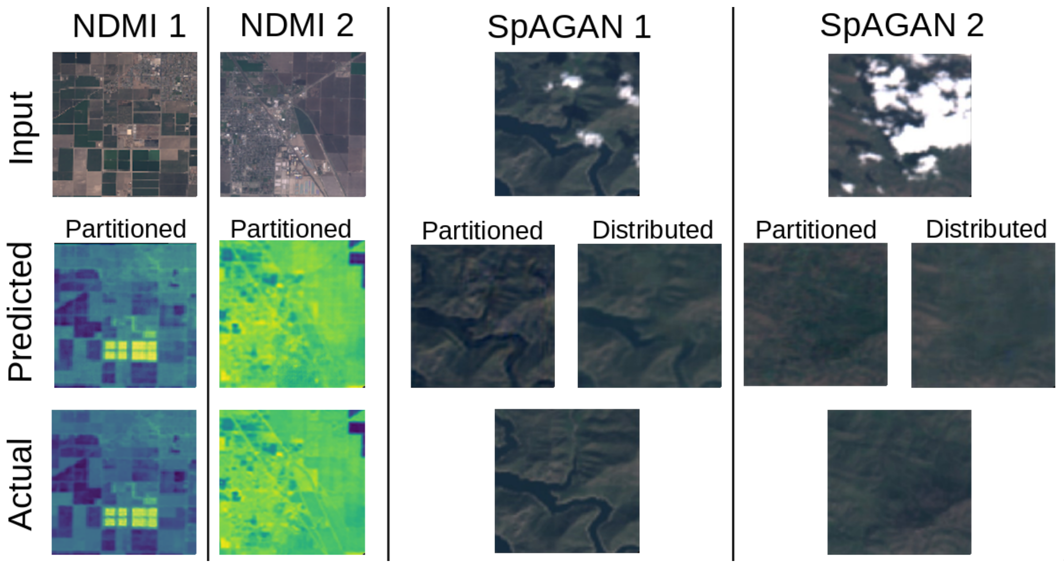


Figure 7.7: Examples of NDMI and SpAGAN predictions visualizing the accuracy of our partition-based training approach.

apply a low-pass filter followed by iterative down-sampling of the image, comparing the luminance, contrast, and structure at multiple scales. This helps in capturing the quality of the image at different resolutions. We see that the **our partitioned solution results in higher accuracy when compared to the distributed approach.**

We see the average PSNR and SSIM for our partitioned training with the SpAGAN model was slightly higher than the distributed training, demonstrating that **the partitioned models generated images more similar to the ground truth image than the distributed model**. Additionally, in Figure 7.7 we see the partitioned models more precisely capture landscape details, such as rivers and mountain ridges, than the distributed model. This is because our partitioned models are able to specialize on a particular region in contrast to the distributed model.

7.2.4 Spatial Transfer Learning

Transfer learning is a broadly applied paradigm in deep learning. It involves taking a model trained on one dataset and applying it to a related task, and it serves to produce more accurate models in a shorter time, with significantly reduced resource requirements. We posit that regions with similar textural qualities will benefit from transfer learning.

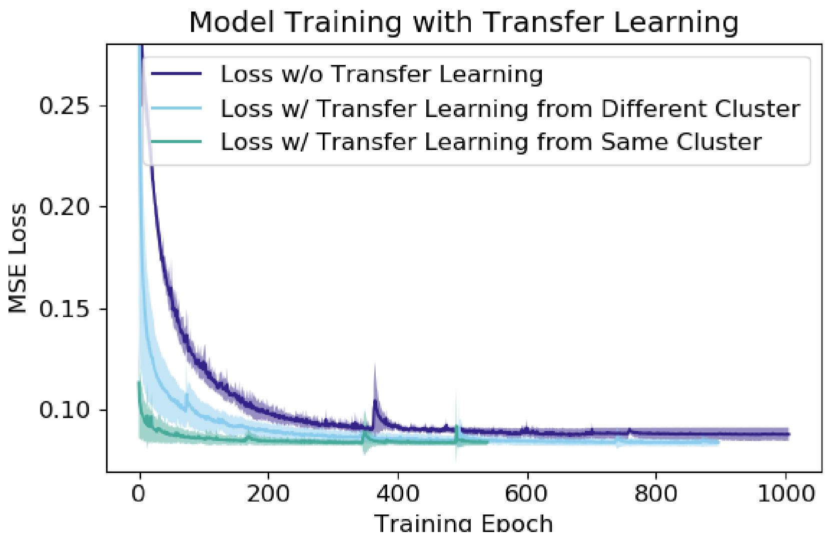


Figure 7.8: The effects of transfer learning on training the NDVI model.

Our solution applies the gray level cooccurrence matrix (GLCM) algorithm to identify similar regions based on geocode bounds. GLCM processes gray-scaled image pixels to produce a matrix capturing neighboring pixel relationships. It is a widely accepted practice to quantify textural qualities of an image. Subsequent analyses are often facilitated by calculating a variety of features on the GLCM; these include dissimilarity, homogeneity, contract, energy, ASM, and correlation.

Given textural features for each geohash we quantify spatial region similarity using the k-means clustering algorithm. GLCM feature matrices are quite large, in our application each feature produces 3 unique 255 x 255 byte structures. To improve the accuracy of k-means, which typically struggles with large-dimensionality data, we compute averages over the main matrix axes. The result is a vector of 18 values, each of which captures qualities of a single GLCM feature. Finally, we dissected the elbow in the cluster silhouette score plot (using multiple distance functions like cosine, Manhattan, and euclidean) to find the ideal cluster count at 12.

In Figure 7.8 we observe that transfer learning has a significant effect on NDVI model training. We partitioned geohash regions into 12 clusters using the aforementioned approach. Transfer learning from a random cluster (naive transfer learning) converges twice as fast as without transfer learning, as well as reaching a lower loss. **Transfer learning from the same cluster resulted in approximately 4 times faster convergence and even lower loss.**

Chapter 8

Conclusions

In this dissertation we have presented a methodology to facilitate efficient storage and analytics over spatiotemporal data. We address scaling to endure several challenges inherit in spatiotemporal workloads, although these solutions may span domains. Specifically, we have demonstrated the breadth of our approach on feature-based data and satellite imagery. These include effectively partitioning and indexing spatiotemporal datasets, scaling systems by distributing over a cluster of machines, interoperation with existing frameworks, support for diverse analytical operations, and efficient model training. This effort has been guided by careful consideration of the following broad research objectives:

1. Building systems which can scale to cope with increases in dataset volumes and diversity.
2. Improving performance, while maintaining accuracy, of analytics through more effective utilization of CPU / GPU, memory, and disk and network I/O.
3. Developing solutions which integrate into existing workflows.

Addressing these challenges requires iterative innovation as the volumes and diversity of spatiotemporal data continue to evolve. However, this work proposes several novel improvements in the current realm of spatiotemporal data storage and analytics.

We have developed systems that scale out – effectively distributing storage and analytics. We load-balance data distribution by employing distributed hash tables (DHTs), data blocks, and spatial image partitions. Our solutions have been evaluated on a diverse collection of datasets for both accuracy and performance. These data range to 100's of terabytes which consist of (1) multi-dimensional feature-based data scaling to over 100,000 features and (2) hyperspectral satellite imagery with varying formats, resolutions, and spatial reference systems.

We demonstrate performance and accuracy over a diverse collection of spatiotemporal operations. This entails application of data sketching algorithms to facilitate in-memory analytics, mitigating several orders of magnitude degradation in disk access performance. The application of spatiotemporally indexed data blocks ensures targeted, sequential disk access. Dispersion-informed data replication provides both dispersion and collocation of spatiotemporal extents, facilitating efficient parallel and local processing. Additionally, we extended the Apache Spark framework to support assorted spatiotemporal functionally encompassing points, lines, and polygons, which benefit from the aforementioned performance improvements.

Our solutions interoperate with popular distributed frameworks, demonstrating the ability to integrate our solutions into existing workflows. We have developed storage systems which communicate using Hadoop Distributed File System (HDFS) native protocols. These expose in-memory sketched datasets and spatiotemporal indexed data blocks with URL-based spatiotemporal queries. Our extensions to Apache Spark serve as an example, seamlessly coordinating with our distributed storage systems. Finally, we employ popular machine learning frameworks to effectively train models over our spatiotemporal data stores.

In summary, the contributions of this work include: (1) integration of data sketching algorithms into distributed storage and analytics frameworks to provide in-memory performance over large data, (2) effective dispersion-informed data replication to enable simultaneous distribution and collocation of spatiotemporal extents, mitigating I/O in data movements during analytics, (3) solutions which adhere to open-sourced, community-driven protocols facilitating seamless interoperation with existing frameworks, (4) integration of spatiotemporal operations with low-level optimizations into existing distributed analytics suites, and (5) novel spatiotemporal deep learning schemes which offer performant workload orchestration and improvements in model evaluations at scale.

8.1 Future Work

We are proud of the innovations addressing interesting, open research questions presented in this dissertation. However, storage and analytics over spatiotemporal data is a continually evolving challenge. As the data evolves, so must solutions for coping with the increasing volumes and diversity. As such, our solutions spark plenty of opportunity for future work.

A critical component in our methodology is partitioning the dataspace along spatiotemporal bounds. This serves as the basis for ensuring data locality while distributing workloads. Currently, this process requires statically defined ranges (e.x., spatial geohashes or temporal durations). However, differences in region complexity expose opportunities for performance improvements. We propose support for dynamic variance in spatiotemporal scopes, where lower complexity regions can be stored / analyzed with lower resolutions and higher complexity with higher resolutions. This is particularly interesting given the breadth of application, where improvements range from storage subsystems through high-level analytics.

Approaches for dynamic variance in spatiotemporal scope must overcome many challenges. First and foremost, they must ensure accurate evaluations. Any degradation of analytical correctness is unacceptable. Rather, it must be maintained with resolution reductions. Also, they must cope with changes in region complexity over time. The ability to dynamically identify the complexity of a spatiotemporal extent and evolve with it is paramount. Data changes, structures are built and destroyed, weather shifts, etc, and the approaches need to cope with these changes.

A prime candidate for this scheme is dynamically down-scaling hyperspectral image resolutions. When analyzing the visible spectrum the Mojave Desert, being almost exclusively homogeneous, will exhibit significantly lower variance than say New York City, with a blend of highly industrial, parks and landscaping, and water components. Downscaling images allows (1) simplification of indices, where fewer entries reduces the overall size of the index and (2) reductions in dataset sizes, overall data reductions will coincide with the factor of downscaling.

Data sketching algorithms rely on maintaining summary statistics for a specific spatiotemporal region. We have applied these technologies, which offer significant dataset size reductions, to facilitate in-memory analytics speed over large data. The accuracy of decompressed data relies on successfully clustering observations where the variance between actual and summary statistics is small. Therefore, accommodating larger regions ensures simplification of the indexing structure, reducing dataset query durations and ultimately reducing the size of the sketched data.

Bibliography

[1] Ching-Yung Lin and Shih-Fu Chang. Robust image authentication method surviving jpeg lossy compression. In *Storage and Retrieval for Image and Video Databases VI*, volume 3312, pages 296–307. International Society for Optics and Photonics, 1997.

[2] Xinpeng Zhang. Lossy compression and iterative reconstruction for encrypted image. *IEEE transactions on information forensics and security*, 6(1):53–58, 2010.

[3] Osama K Al-Shaykh and Russell M Mersereau. Lossy compression of noisy images. *IEEE Transactions on Image Processing*, 7(12):1641–1652, 1998.

[4] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[5] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*, pages 366–379. Springer, 2011.

[6] Sriram Lakshminarasimhan, John Jenkins, Isha Arkatkar, Zhenhuan Gong, Hemanth Kolla, Seung-Hoe Ku, Stephane Ethier, Jackie Chen, Choong-Seock Chang, Scott Klasky, et al. Isabela-qa: query-driven analytics with isabela-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[7] Zhengzhang Chen, Seung Woo Son, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Numarck: machine learning algorithm for resiliency and checkpointing. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 733–744. IEEE, 2014.

[8] Zheng Yuan, William Hendrix, Seung Woo Son, Christoph Federrath, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Parallel implementation of lossy data compression for temporal data sets. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 62–71. IEEE, 2016.

[9] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[10] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[11] Jayadev Misra and David Gries. Finding repeated elements. Technical report, Cornell University, 1982.

[12] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[13] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[14] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[15] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *ACM SIGMOD*, pages 775–787. ACM, 2017.

[16] Yufei Tao, G. Kollios, J. Considine, F. Li, and Dimitris Papadias. Spatio-temporal aggregation using sketches. In *Proc. of the Intl. Conference on Data Engineering*, pages 214–225, March 2004.

[17] Thilina Buddhika, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2566, 2017.

[18] Thilina Buddhika, Sangmi Lee Pallickara, and Shrideep Pallickara. Pebbles: Leveraging sketches for processing voluminous, high velocity data streams. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):2005–2020, 2021.

[19] Stefan Lang, Geoffrey J Hay, et al. Geobia achievements and spatial opportunities in the era of big earth observation data. *ISPRS International Journal of Geo-Information*, 8(11):474, 2019.

[20] Nicholus Mboga, Stefanos Georganos, et al. Fully convolutional networks and geographic object-based image analysis for the classification of vhr imagery. *Remote Sensing*, 11(5):597, 2019.

[21] Lu Xu, Dongping Ming, Wen Zhou, Hanqing Bao, Yangyang Chen, and Xiao Ling. Farmland extraction from high spatial resolution remote sensing images based on stratified scale pre-estimation. *Remote Sensing*, 11(2):108, 2019.

[22] Hassan Bazzi, Nicolas Baghdadi, Dino Ienco, et al. Mapping irrigated areas using sentinel-1 time series in catalonia, spain. *Remote Sensing*, 11(15):1836, 2019.

[23] Stefanos Georganos, Tais Grippa, Moritz Lennert, et al. Scale matters: Spatially partitioned unsupervised segmentation parameter optimization for large and heterogeneous satellite images. *Remote Sensing*, 10(9):1440, 2018.

[24] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[25] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[26] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. *International Conference on Very Large Data Bases*, 1987.

[27] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

[28] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, University of Maryland, 1993.

[29] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[30] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2pr-tree: An r-tree-based spatial index for peer-to-peer environments. In *International Conference on Extending Database Technology*, pages 516–525. Springer, 2004.

[31] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal—The International Journal on Very Large Data Bases*, 16(2):165–178, 2007.

[32] Verena Kantere, Spiros Skiadopoulos, and Timos Sellis. Storing and indexing spatial data in p2p systems. *IEEE Transactions on Knowledge and Data Engineering*, 21(2):287–300, 2008.

[33] Geoffrey Fox, Sang Lim, Shrideep Pallickara, and Marlon Pierce. Message-based cellular peer-to-peer grids: foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21(3):401–415, 2005.

[34] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. Towards enabling peer-to-peer grids. *Concurrency and Computation: Practice and Experience*, 17(7-8):1109–1131, 2005.

[35] Yogesh L Simmhan, Sangmi Lee Pallickara, Nithya N Vijayakumar, and Beth Plale. Data management in dynamic environment-driven computational science. In *Grid-based problem solving environments*, pages 317–333. Springer, 2007.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[38] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[40] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[41] Glustre file system project, 2020.

[42] Nusrat Sharmin Islam, Xiaoyi Lu, Md Wasi-ur Rahman, Dipti Shankar, and Dhabaleswar 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*, pages 101–110. IEEE, 2015.

[43] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[44] Jiamin Lu and Ralf Hartmut Güting. Parallel secondo: boosting database engines with hadoop. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 738–743. IEEE, 2012.

[45] Yunqin Zhong, Jizhong Han, Tieying Zhang, Zhenhua Li, Jinyun Fang, and Guihai Chen. Towards parallel spatial query processing for big spatial data. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2085–2094. IEEE, 2012.

[46] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[47] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*, pages 1352–1363. IEEE, 2015.

[48] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 1, pages 7–16. IEEE, 2011.

[49] Dan Han and Eleni Stroulia. Hgrid: A data model for large geospatial data sets in hbase. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 910–917. IEEE, 2013.

[50] Ningyu Zhang, Guozhou Zheng, Huajun Chen, Jiaoyan Chen, and Xi Chen. Hbasespatial: A scalable spatial data storage based on hbase. In *2014 IEEE 13th international conference on trust, security and privacy in computing and communications*, pages 644–651. IEEE, 2014.

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

[52] Regina Obe and Leo Hsu. Postgis in action. *Geoinformatics*, 14(8):30, 2011.

[53] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, volume 9473, page 94730F. International Society for Optics and Photonics, 2015.

[54] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

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

[56] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[57] Apache spark project, 2020.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[59] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[60] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query processing of massive trajectory data based on mapreduce. In *Proceedings of the first international workshop on Cloud data management*, pages 9–16. ACM, 2009.

[61] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial queries evaluation with mapreduce. In *2009 Eighth International Conference on Grid and Cooperative Computing*, pages 287–292. IEEE, 2009.

[62] Kai Wang, Jizhong Han, Bibo Tu, Jiao Dai, Wei Zhou, and Xuan Song. Accelerating spatial data processing with mapreduce. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 229–236. IEEE, 2010.

[63] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. IEEE, 2009.

[64] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Proceedings of the 15th international conference on extending database technology*, pages 38–49. ACM, 2012.

[65] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012.

[66] Afsin Akdogan, Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. Voronoi-based geospatial query processing with mapreduce. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 9–16. IEEE, 2010.

[67] Ahmed Eldawy, Yuan Li, Mohamed F Mokbel, and Ravi Janardan. Cg_hadoop: computational geometry in mapreduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 294–303. ACM, 2013.

[68] Ahmed Eldawy and Mohamed F Mokbel. Pigeon: A spatial mapreduce language. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1242–1245. IEEE, 2014.

[69] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[70] Furqan Baig, Hoang Vo, Tahsin Kurc, Joel Saltz, and Fusheng Wang. Sparkgis: Resource aware efficient in-memory spatial query processing. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 28. ACM, 2017.

[71] Jia Yu, Jinxuan Wu, and Mohamed 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, page 70. ACM, 2015.

[72] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13):1565–1568, 2016.

[73] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1071–1085. ACM, 2016.

[74] HaiYing Wang, Rong Zhu, and Ping Ma. Optimal subsampling for large sample logistic regression. *Journal of the American Statistical Association*, 2017.

[75] Rong Zhu, Ping Ma, Michael W Mahoney, and Bin Yu. Optimal subsampling approaches for large sample linear regression. *arXiv preprint arXiv:1509.05111*, 2015.

[76] Rong Zhu. Poisson subsampling algorithms for large sample linear regression in massive data. *arXiv preprint arXiv:1509.02116*, 2015.

[77] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[78] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[79] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[80] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. Borderline-smote: a new over-sampling method in imbalanced data sets learning. In *International Conference on Intelligent Computing*, pages 878–887. Springer, 2005.

[81] Chumphol Bunkhumpornpat, Krung Sinapiromsaran, and Chidchanok 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*, pages 475–482. Springer, 2009.

[82] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[83] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

[84] Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. A divide-and-conquer solver for kernel support vector machines. In *International Conference on Machine Learning*, pages 566–574, 2014.

[85] Qi Guo, Bo-Wei Chen, Feng Jiang, Xiangyang Ji, and Sun-Yuan Kung. Efficient divide-and-conquer classification based on feature-space decomposition. *arXiv preprint arXiv:1501.07584*, 2015.

[86] Yingjie Tian, Xuchan Ju, and Yong Shi. A divide-and-combine method for large scale nonparallel support vector machines. *Neural Networks*, 75:12–21, 2016.

[87] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231, 2012.

[88] Martín Abadi, Paul Barham, Jianmin Chen, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[89] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *Conference on Neural Information Processing System*, 2017.

[90] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[91] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[92] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

[93] D. Jacob, D. Polani, and C. L. Nehaniv. Legs that can walk: embodiment-based modular reinforcement learning applied. In *2005 International Symposium on Computational Intelligence in Robotics and Automation*, pages 365–372, 2005.

[94] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175, 2017.

[95] Sooraj Bhat, Charles L Isbell, and Michael Mateas. On the difficulty of modular reinforcement learning for real-world partial programming. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 318. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[96] Kevin Bruhwiler, Paahuni Khandelwal, Daniel Rammer, Samuel Armstrong, Sangmi Lee Pallickara, and Shrideep Pallickara. Lightweight, embeddings based storage and model construction over satellite data collections. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 246–255. IEEE, 2020.

[97] Paahuni Khandelwal, Daniel Rammer, Shrideep Pallickara, and Sangmi Lee Pallickara. Mind the gap: Generating imputations for satellite data collections at myriad spatiotemporal scopes. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 92–102. IEEE, 2021.

[98] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[99] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. imapreduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, 2012.

[100] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, pages 135–146. ACM, 2010.

[101] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[102] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[103] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[104] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 1, page 3, 2014.

[105] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: a new platform for distributed machine learning on big data. *Big Data, IEEE Transactions on*, 1(2):49–67, 2015.

[106] Daniel Rammer, Walid Budgaga, Thilina Buddhika, Shrideep Pallickara, and Sangmi Lee Pallickara. Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 468–477. IEEE, 2018.

[107] Daniel Rammer, Sangmi Lee Pallickara, and Shrideep Pallickara. Atlas: A distributed file system for spatiotemporal data. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 11–20, 2019.

[108] Chi-Ming Lin and Mitsuo Gen. Multi-criteria human resource allocation for solving multistage combinatorial optimization problems using multiobjective hybrid genetic algorithm. *Expert Systems with Applications*, 34(4):2480–2490, 2008.

[109] Maria João Alves and Marla Almeida. Motga: A multiobjective tchebycheff based genetic algorithm for the multidimensional knapsack problem. *Computers & operations research*, 34(11):3458–3470, 2007.

[110] Protocol Buffers. Google's data interchange format, 2011.

[111] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[112] Daniel Rammer, Thilina Buddhika, Matthew Malensek, Shrideep Pallickara, and Sangmi Pallickara. Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines. *IEEE Transactions on Big Data*, 2019.

[113] Daniel Rammer, Sangmi Lee Pallickara, and Shrideep Pallickara. Towards timely, resource-efficient analyses through spatially-aware constructs within spark. In *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing*, 2020.

[114] Location tech jts project, 2020.

[115] Ram Sriharsha. Geospatial analytics using spark.

[116] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment*, 11(11):1661–1673, 2018.

[117] Daniel Rammer, Kevin Bruhwiler, Paahuni Khandelwal, Sam Armstrong, Shrideep Pallickara, and Sangmi Lee Pallickara. Small is beautiful: Distributed orchestration of spatial deep learning workloads. In *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing*, 2020.

[118] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[119] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.