2024 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT) | 979-8-3503-6730-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/BDCAT63179.2024.00047

SCRYBE: Enabling Programmatic Interfaces for Explorations Over Voluminous Spatiotemporal Data Collections

Kassidy Barram*, Sangmi Lee Pallickara*, and Shrideep Pallickara*
*Department of Computer Science, Colorado State University, Fort Collins, CO, USA
Email: kbarram, sangmi, shrideep@colostate.edu

***Abstract*—This study focuses on enabling programmatic interfaces to perform exploratory analyses over voluminous data collections. The data we consider can be encoded in diverse formats and managed using diverse data storage frameworks. Our framework, code named SCRYBE, manages the competing pulls of expressive computations and the need to manage resource utilization in shared clusters. The framework includes support for differentiated quality of service allowing preferentially higher resource utilization for certain users. We have validated our methodology with voluminous data collections housed in relational, NoSQL/document, and hybrid storage systems. Our performance benchmarks profile several aspects of our methodology, and demonstrate the effectiveness of our methodology.**

***Index Terms*—big data, programmatic interfaces, notebooks, containers, orchestration engines, data analysis.**

I. INTRODUCTION

Datasets continue to be made available at increased precision, resolution, and frequency in several domains. Given the data volumes, it is infeasible for users to create their own personal copies of the data. Often an organization may have datasets with curtailed redistribution rights that might preclude downloads. The class of data that we consider in this study are spatio-temporal i.e. the data items (numeric, categorical, or ordinal) have spatial – georeferenced using $< latitude, longitude >$ coordinates and temporal dimensions associated with it. Further, the data may be stored in myriad encoding formats.

Users are interested in exploring these datasets but often encounter several headwinds. Dominant approaches to enabling explorations may be broadly classified into three themes: enabling visualizations, leveraging web services or service oriented architectures, and allowing raw data downloads. To manage development, data access overheads, and preserve interactivity, visualizations are typically limited to a predefined set of visual analytic operations supported within the tool. Systems based on service oriented architectures expose a set of services that are performed server side on demand. Some systems allow raw data downloads which require the user to perform the task of identifying the backend storage systems, perform indexing operations, and express operations using either queries or launching server-side tasks. Raw data downloads can become infeasible as data volumes and the number of encoding formats increase. Some frameworks allow users to launch complex server-side jobs, but these are often restricted to users within the organizations. Finally, details regarding data storage frameworks, schemas, and encoding formats alongside any indexing schemes are often opaque to the user.

In this study, we explore enabling programmatic interfaces to server-side data collections. There are several advantages to doing this. First, it precludes the need for capital expenses for users. Second, users can identify and explore the types of operations and analysis that are of interest. Users can design expressive analysis tasks that are no longer confined to the set of tasks exposed in visualization engines or services. Crucially, users can supplement their own processing logic with those available in libraries, etc. Simplified programming interfaces also allow users to express operations on data that are inherently simpler than launching them on raw datasets.

A. *Challenges*

There are several challenges in enabling programming interfaces to data-driven explorations over voluminous datasets.

1) Incorrect logic (runaway recursions, cascading data accesses) and the complexity of the processing logic, data accesses, and network data shuffles can have adverse server-side implications.
2) Access restrictions need to be preserved with write-operations being disallowed. Furthermore, locking mechanisms need to be lightweight and fine-grained to preclude lockout situations.
3) Ensuring that resources server-side aren't impacted. Since computations triggered by programmatic explorations execute within shared clusters, a key challenge is to ensure that server-side utilization of resources is regulated.

B. *Research Questions*

The overarching objective of this study is to enable programmatic interfaces to backend data stores. Specific research questions that we explore include:

RQ-1: How can we support explorations over voluminous datasets stored in a diversity of formats and using different storage frameworks?

RQ-2: How can we support effective visualizations/explorations? Users are keen on visualizing the outcomes of data

979-8-3503-6730-0/24/$31.00 ©2024 IEEE
DOI 10.1109/BDCAT63179.2024.00047

processing tasks and mechanisms need to be in place to ensure timeliness of operations.

RQ-3: How can we ensure that data explorations do not overwhelm server-side resources? Because the data collections hosted server-side operate in shared clusters, we must ensure that collocated users and tasks in shared clusters are not adversely impacted.

RQ-4: How can we include support for differentiated QoS (quality of service)? In particular, the objective is to ensure that different users have access to different sets of capabilities.

C. Approach Summary

Our framework, named SCRYBE, provides a programmatic interface to voluminous spatio-temporal data collections via Jupyter Notebooks hosted in a web-browser. We used Python as the starting point because it represents a rich ecosystem (numpy, pandas, scikit-learn, etc.) of libraries for performing data wrangling, analysis, and model fitting operations. All of these are immediately available to the user. Finally, notebooks are also shareable. This simplifies collaborative activities as opposed to sharing scripts, libraries, etc.

To ensure that notebooks individually and collectively do not breach resource thresholds server-side, we leverage containers and orchestration engines. Specifically, the orchestration is performed by Kubernetes and the containers are built using Docker. We leverage Kubernetes to provide system-wide resource utilization metrics collected via Prometheus and Kubernetes ResourceQuotas to ensure thresholds are not breached.

Our data accesses and retrievals reconcile the complexity of dealing with a diversity of backend storage systems, authorization, schemas, and access controls. We provide a simplified interface to the data, and individual queries are programmatically composed. We use a lightweight query refinement scheme that reorganizes the query predicate based on identification of features that are indexed (either individual or compound indexes) by the storage framework, and also leveraging heuristics to identify predicates that prune the search space.

Rather than provide access to all records that satisfy the query, we provide access to a representative sample. This allows us to limit the number of I/O retrievals and network transmissions that are performed server-side. All data analysis operators are performed on the sample. This has two advantages. It conserves resources server-side and ensures timely completion of operations at the client.

We support a curated set of analysis operators that can be categorized as sampling, graphing, and visualizations. For the more resource intensive visualizations, we leverage client-side GPU accelerations for rendering, which also alleviates load server-side.

We support differentiated services based on roles. Users belonging to certain groups, based on their roles, are provided enhanced access to capabilities. These include higher resource thresholds, access to a greater number of collections and operators, and increased sample sizes. All differentiated QoS work within the confines of the aggregated resource thresholds that are configured for SCRYBE.

D. Paper Contributions

SCRYBE facilitates programmatic explorations of voluminous spatio-temporal datasets that are housed in shared clusters. Key contributions of our framework include:

- A simplified programmatic composition of queries that abstracts away the query semantics expected by the data storage that manages the underlying collection. Query transformations are handled opaquely by the backend storage framework.
- Our methodology is independent of the underlying storage system. We have validated the suitability of SCRYBE with storage systems based on relational storage (Postgres), NoSQL (MongoDB), and semi-structured/hybrid systems (Druid/HDFS).
- Support for differentiated QoS that allows role-based access to additional computing, datasets, and server-side resources.
- The framework includes a novel mix of sampling, streaming, query interfaces, and containers to manage the competing pulls of expressiveness and timeliness without overwhelming server-side resources.
- Our lightweight query refinement scheme reorganizes query predicates to produce equivalent queries that execute faster while accounting for indexing and pruning of the search space.
- Support for differentiated QoS in resource limited environments.

E. Paper Organization

The remainder of the paper is organized as follows. Section II outlines background and related work. Section III describes several key aspects of our methodology and system architecture. Section IV includes a discussion of our performance benchmarks and profiling. Finally, section V outlines our conclusions and future work.

II. RELATED WORK

Interactions with voluminous data are typically facilitated using methods that can be broadly categorized as being based on (1) visualization interfaces, (2) service-oriented architectures, (3) computational frameworks that manage submission of processing tasks that operate on portions of the dataspace. Each of these approaches view the data accesses and processing aspects from different vantage points and requirements spanning latencies, resource thresholds, and expressivity of computations.

Visualization interfaces: Several systems, such as Tableau [1], QGIS [2], provide a visualization-driven interface to the datasets. These interfaces facilitate curated, preset explorations of the data space based on pivots, panning, drilldowns, and rollups across the feature space. A key objective of these methods tends to be interactivity and visual artifacts that facilitate navigability. Often such systems rely on computing

composite overviews in their data representations to minimize data accesses server-side. While SCRYBE includes support for visualization via its support for GPU-accelerated choropleth maps and charting, the core focus is on a programmatic interface that provides users with control over the expressivity of their analytic tasks.

Service oriented architectures: Systems based on service oriented architectures [3], [4] encapsulate accesses to data via preconfigured functions. In the case of web services, these endpoint configurations were accomplished using the web services description language [5], that provided language-agnostic bindings to backend capabilities. REST (representational state transfer) based systems [6] rely on stateless communications between the endpoints to accomplish tasks. In RESTful systems, every aspect of the task that needs to be accomplished on the server-side must be encapsulated as parameters that are included as part of the invocation. This allows RESTful systems to seamlessly interact with multiple clients concurrently in a language-agnostic fashion. Remote procedure calls and distributed object-based systems such as CORBA, RMI, and .NET-based systems are the original precursors of such architectures. SCRYBE, with its programmatic interfaces to such datasets, can be viewed as complementary to these efforts.

Computational frameworks: Cloud computing frameworks have been used to process voluminous datasets. Apache Hadoop, originally developed at Yahoo, is the most widely used implementation of the MapReduce framework [7], [8]. Hadoop supports several applications at Yahoo and is also hosted within Amazon's EC2 cloud [9]. Microsoft Research's Dryad, based on directed, acyclic graphs (DAG), represents computations as sequential programs connected using one-way channels [10]. Traditional high-throughput computing systems have been used to support data driven execution of execute-once tasks using DAG task graphs [11]. DAGMan [12], Karajan [13], Swift [14], and VDS [15] rely on batch schedulers to execute parallel tasks and are unsuitable for processing streams in real time because of the high overhead required to schedule and dispatch tasks. Glide-in approaches rely on multitiered scheduling [16]–[19] with the second-tier scheduler dispatching tasks to resources allocated using first-tier traditional batch schedulers. SCRYBE differs from these approaches in its focus on leveraging sampling, support for multiple backend data stores (relational, NoSQL, or hybrid) and the need to preserve resource utilization thresholds within the cluster.

Virtualization and dissemination: Efforts have explored modeling [20] and performance considerations [21] in the migration of multi-tier applications to the cloud. Such frameworks also rely on effective messaging infrastructures [22] while accounting for security considerations [23].

SCRYBE complements the aforementioned efforts with its focus on programmability, preservation of resource utilization thresholds, alleviating server-side resource contentions, a notebook interface that simplifies incremental updates and collaboration, charting, differentiated QoS, and user-driven expressivity in the programming logic for computations.

III. METHODOLOGY

Our methodology to provide an effective, resource-aware, and accessible programmatic interface to a diverse set of backend capabilities encompasses several elements, including: (1) designing a containerized infrastructure to orchestrate workloads, (2) sampling schemes to alleviate data processing requirements, (3) supporting datastore-agnostic, simplified query semantics, (4) enforcing resource thresholds on the server-side, (5) leveraging client-side resources, and (6) support for differentiated QoS.

A. *High-level System Overview*

We have developed SCRYBE to be robust, scalable, and dynamic to support diverse workloads and a variable number of users while preventing over consumption of resources and encapsulating privileges. This is underpinned by two components: client containers and server containers. All containers in SCRYBE are built using Docker [24]. A high-level diagram of our system is shown in Fig.1.

The client containers are served via the JupyterHub framework, which allows for a configurable Jupyter Notebook environment to be hosted in a browser [25]. Each user container is prepopulated with examples and tutorials for how to use our service. We also define Python classes that are used by the user to interface with our backend services and utility classes which our system uses to gather user information. Both are stored in a user inaccessible location within the container so that users cannot change or alter values to achieve behavior not specifically provided by SCRYBE.

We built our servers in Java. Java was chosen because of its performance improvements over Python and its Object Oriented nature have allowed us to use Software Engineering principles to minimize code duplication and have increase maintainability. By containerizing our server functionality, we can have multiple instances of the server that are running across the cluster. Besides enabling load balancing, this dispersal of workloads increases fault-tolerance because the system remains functional even if a given machine were to fail. Containerization additionally allows for a configurable number of servers, which can increase or decrease dependent on the load the system is experiencing.

B. *Sampling Schemes* [RQ-1, RQ-3]

Given the voluminous nature of our datasets, it is infeasible to provide users access to the entire dataset. Instead, we have developed sampling schemes that are representative of the data characteristics desired by the user and produce a commensurate reduction in network and disk I/O. To use our schemes, a user specifies a dataset name and field and the data returned will be sampled according to the predefined function they use.

The first query that we have designed is representative of the distribution of numeric values. The user specifies the dataset name and a numeric field of interest, this is then

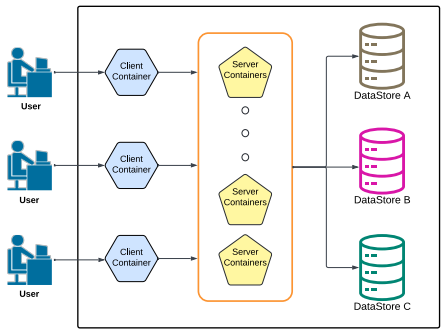

Fig. 1. SCRYBE'S system architecture was designed and developed to scale in order to support diverse workloads. Upon sign-in, a client container is created within the cluster and assigned to the user. User queries are then issued to a cluster of server containers. The number of server containers vary dependent on system load. The server containers then route requests to the correct data store. This request resolution is opaque to the user.

sent to the server where we cluster and bin the distribution of the values; next, we sample equally from each bin. To create the bins, we leverage a dynamic programming approach to Kmeans clustering called Ckmeans [26]. We additionally support a form of stratified sampling if the user is interested in sampling with respect to categorical values. Stratified sampling is used when the data can be partitioned by subpopulation. This technique takes a dataset and a categorical field, a sample is then created by determining the proportional representation of the data per category.

An additional sampling scheme has been designed specifically for the type of data that is most commonly represented within our data stores. This data is spatio-temporal, where the data is geocoded either with a latitude/longitude pair or a hierarchical code representing a geospatial region and a time stamp. Because of this spatial component in our data, we support a sampling scheme that samples the spatial extents proportional to the total number of observations available for that region. This technique preserves the distribution of the data points geographically. For all sampling techniques that we implement, if there are too few observations to be sampled based on our threshold all observations are returned. The accessibility of this sampling technique is dependent on the operations supported by the underlying database framework. Specifically, to support this sampling technique, regional polygonal queries must be supported.

In addition to our bespoke spatial sampling schemes, we support random sampling. This leverages the built-in random sampling operations of the underlying database framework. From a given dataset, this technique samples data values such that each has an equal probability of being selected. This sampling technique is not guaranteed to be representative of the distribution of the data, which underpins the necessity for the other supported techniques.

C. *User Environment & Simplified Query Semantics* [RQ-2]

In SCRYBE, programmatic explorations are performed using Python and Jupyter Notebooks. Three key factors informed our design decision. First, Python is one of the most preferred languages for data processing, wrangling, and analytic tasks. A recent (2023) Stack Overflow developer survey revealed that 83.7% of data science professionals use Python [27]. Second, there is a rich ecosystem of libraries for scientific computations, data fitting algorithms, and myriad language bindings. This allows users to supplement their analysis in substantive ways. Finally, Jupyter Notebooks provide a rich framework for creating, sharing, and incremental refinements of updates. The notebook interface also allows us to harness user familiarity and reduce barriers to entry for new users to SCRYBE.

A key requirement for enabling programmatic explorations is the creation of a curated user environment that is intuitive and easy-to-use —especially for performing analytics that the user may be interested in. To achieve this goal, we have created several tutorial notebooks to provide step-by-step examples of all capabilities that are offered in the service. These notebooks are prepopulated within a user's environment when their container is allocated upon service launch. However, the user is not limited to the examples we provide. They are able to create files, Jupyter Notebooks, save, delete, and download all materials that are defined within their home directory. Because the system resides within a cluster where the user accesses via a browser, the user is able to immediately begin using the system without the need to configure the environment. These features allow the user experience to be streamlined.

We have created simplified querying semantics such that the user is able to perform queries without needing to know the particulars of the underlying data stores or services that are being used. These capabilities are possible by leveraging metadata that is stored in an inaccessible location within each container that is allocated to the user. This metadata performs query resolution such that dataset names, when specified in a query, are resolved to a data source. Additionally, queries that return data to the users have a standardized format regardless of the data source they originated from and the storage format used therein. This allows the users to perform data collations across different data stores seamlessly despite their disparate storage formats. We chose Pandas DataFrame [28] for our data because it allows users to have immediate access to the most common packages and libraries used in the Python language in addition to the capabilities we provide.

To further simplify, we implement a programmatic entry point called the ClientGateway. This entry point abstracts all client/server complexity away from the user. This allows the user to call functions on an instance of the ClientGateway which then formats their input, connects to the backend server, and then formats the response. We have chosen this mechanism because our system uses gRPC (Google Remote Procedure Calls) to facilitate client - server communication. The gRPC framework was chosen because it is efficient, high performance, and supports communications between clients and servers written in different languages [29]. This was crucial for our system because our clients are Python based, but our servers are written in Java. However, gRPC is a frame-

work which has complex syntax and a different conceptual model than the typical request/response mechanism that most users are familiar with. For this reason, a class to abstract complexities was necessary to allow the user a simple and accessible interface to communicate with the backend services.

D. *Enforcing Resource Thresholds* [RQ-3]

One of the main challenges of a system that allows users access to cluster resources is ensuring that users are not allowed more than their fair share of shared system resources. Because data intensive accesses and the corresponding residency requirements put strain on server-side resources, we focus primarily on minimizing over consumption of memory resources. We approach this from two perspectives: the total memory and CPU resources used by all SCRYBE processes and individually by the user.

To prevent the total sum of SCRYBE resources from overwhelming the cluster, we leverage our cluster orchestration framework Kubernetes [30]. We use the Kubernetes Namespace and ResourceQuota objects to facilitate and enforce cluster thresholds. A Kubernetes Namespace allows for isolation of a group of resources from the rest of the cluster. A ResourceQuota is a mechanism that allows for preventing the total sum of all resources in a Namespace to not surpass configured limits. Because of the containerized nature of our user environments and our servers, we are able to assign both to a Namespace with a ResourceQuota which prevents users from being able to collectively surpass resource limits.

We implement memory and CPU usage thresholds on the client-side using endogenous and exogenous mechanisms. The endogenous mechanism takes user memory usage information from within the user's notebook container to allow or disallow the user from querying for more data from the server-side. The memory usage information is determined by performing a lookup of the memory usage statistics that are maintained in the underlying Linux file system when a user performs a query. This memory information is then used on the server to determine how much data the user is able to receive. This mechanism is completely stateless, so the correct amount of data is returned to the user regardless of which server instance a user queries. The exogenous mechanism is implemented by leveraging Kubernetes pod limits. Limits prevent the user from using resources that are beyond the specified limit amount for a container.

E. *Leveraging Client Side Resources* [RQ-2, RQ-3]

In order to distribute the computational load, we push computations to the client whenever possible. We leverage the client to format data and to visualize analytics. For response formatting, after a query, the client receives a gRPC stream of strings, it then collects the streamed results and formats them into a Pandas Dataframe. The client dynamically resolves different response formats from the disparate data stores based on metadata lookup at the beginning of the query, where it then returns the standardized format.

For visual analytics within our system, we provide several built-in visualization techniques. These visualizations rely on data from within our cluster to visualize maps, charts and graphs on the client. These visualizations are rendered using the CPU and GPU of the client. To further utilize client-side resources, we built our mapping visualizations using PyDeck. PyDeck is a Python implementation of Deck.gl, which pushes graphical rendering to the client's GPU [31]. The Choropleth mapping visualization (for identifying spatial variation of data) is the most resource intensive visualization we implement. Leveraging GPU resources at the client-side allows us to reduce resource utilizations on the server-side.

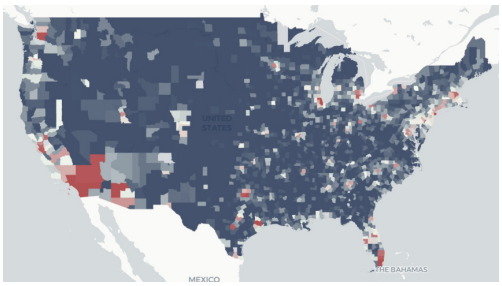

Fig. 2. SCRYBE's built-in mapping capabilities for spatial data. The data being visualized is the Social Vulnerability Index per county. We developed this mapping feature using a framework that pushes computation to the user's GPU. This helps disperse workloads from the backend infrastructure to the client-side.

Average rendering time for basic graphs from query to visualization took 3.01s and utilized the GPU for 27.6ms. With our choropleth visualization, the query took 4.31s and utilized 106ms of the GPU (Table I). The PyDeck visualization is more complex, and incurs additional network costs for its coloring operation than the other basic graphs and is 30.2% slower. However, the Pydeck visualization **leverages the user's GPU 73.9% more than the basic graphs**. This result is particularly important because of the geospatial nature of our data that requires choropleth mapping.

TABLE I
For our most complex visualization technique, choropleth mapping, we specifically leverage a framework that pushes rendering onto the user's GPU.

Graph Type	Time to Render	GPU Utilization
Basic	3.01 s	27.6 ms
Choropleth	4.31 s	106 ms

F. *Enabling a Rich Ecosystem of Data Explorations* [RQ-1]

In order to make data exploration and analysis easy and intuitive for the user, we have developed several built-in capabilities that interface with our backend infrastructure. The set of capabilities we have implemented have been chosen in order to allow the users to glean preliminary insights into the data before launching resource intensive analytics. We have implemented Random Forest Feature Importance, visible in figure Fig. 5. This allows the user to determine and rank the most crucial features. This allows users to prune

(less important) features during machine learning based model fitting operations while reducing memory and computational costs during training.

To visualize the distribution of the data, we have incorporated support for several constructs. For spatial data, we have implemented choropleth maps to render spatial variations for a feature of interest. As shown in Fig. 2, the distribution of the data geographically for a feature is visualized. For numeric and categorical data, we implement several techniques: scatter plots, violin plots, box and whisker plots, and KDEs (kernel density estimation). The KDE shows the smoothed distribution of values using a sample of the data. An example of KDE can be seen in Fig. 4. For exploring time series data, we have implemented the statiscal models ARIMA (Auto-Regressive Integrated Moving Average) and its seasonal implementation, SARIMA. This allows the user to analyze trends in the time series data and predict future values. An example of the violin plots we provide is shown in Fig. 3.

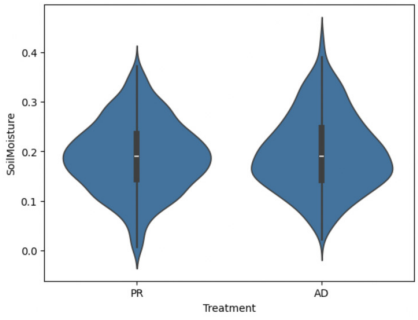

Fig. 3. In this figure, we have visualized the distribution of soil moisture relative to soil treatment using a violin plot. This built-in visualization allows the user to explore relationships between numeric data and categorical data.

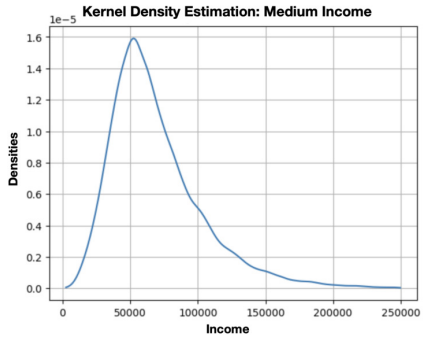

Fig. 4. Our methodology supports a diverse set of built-in data exploration techniques to make our application easy and accessible for a broad set of users. Above we illustrate the visualization of the Kernel Density Estimation of median income. Along the Y-axis we show the densities, the X-axis indicates the range of incomes represented in the data.

G. *Support for Differentiated Services* [RQ-4]

To support differentiated QoS, we leverage RBAC (role based access control). This allows our system to control usage of computational, network, and memory resources as well as controlling access to data stores and datasets, based on a user's identity. We implement RBAC on the client and server-side. The client-side implementation is built into the metadata lookup that resolves queries to data stores. This prevents lower privilege users from accessing data stores that are more resource intensive to query and certain sensitive datasets. This operation resides on the client to reduce the necessary network traffic to perform this lookup on the server-side.

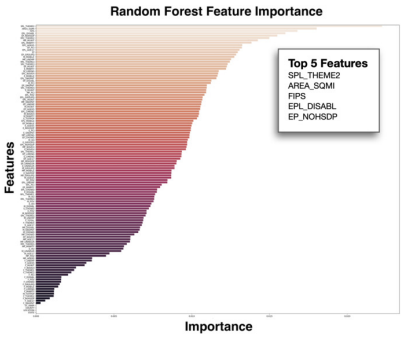

Fig. 5. Built-in visualization of the Random Forests Feature Importance that allows users to perform preliminary data explorations prior to expensive model training operations.

The server-side RBAC we have implemented uses information that is included in a user query when a request is issued on the client. This addition of user information is completely opaque to the user and is abstracted away by our implementation. Each query contains memory resource utilization in the container, username, and requests per minute. On the server, this information is then used to determine how much data the user can receive dependent on their QoS.

We utilize 2 mechanisms: sample size scaling and rate throttling. The sample size is determined using equation (1). b represents initial sample size (based on user's QoS), M is initial memory usage of a client container, x is active memory usage, and Q is a QoS scaling factor. This, relative to memory usage, will decrease the sample size a user is able to receive for a given dataset. The QoS scaling factor will control the rate at which a user's sample size decreases. The behavior of this function is visible in Fig. 7.

$$sampleSize = b \cdot \left(\frac{b}{b - \ M \ + \ \left(\frac{x}{Q}\right)} \right) \tag{1}$$

User's request rate is throttled using the equation (2) below:

$$\frac{sampleSize}{\left\lceil \frac{requestsPerMinute}{qosScalingFactor} \right\rceil} \tag{2}$$

Due to the QoS Scaling Factor, a user is able to issue more requests per minute than a user with a lower QoS before their sample size is reduced. This regulating behavior of SCRYBE is demonstrated in Fig. 6.

IV. PERFORMANCE BENCHMARKS

Our empirical benchmarks profile several aspects of our methodology. In particular, we profile the efficacy of con-

tainerized and distributed backend servers compared to a implementation executed directly on a host machine. We additionally explore the impact of our endogenous and exogenous thresholding schemes on resource utilization within the cluster. Finally, we determine the interactivity of our data exploration techniques from the user's perspective.

A. *Experimental Setup*

Our experiments were performed in a cluster of 62 machines, 50 with an 8-core CPU running at 2.10GHz, 64 GB of DDR3 RAM, and 5400RPM hard disks. A subset of 12 machines have upgraded CPU and RAM with 16-core CPU, running at 3.6GHz, and 98GB of RAM.

We utilized 3 datastores with diverse configurations and system architectures to validate our methodology. We used a replicated and sharded MongoDB cluster, an installation of a distributed Apache Druid database, and a PostGres database deployed using Kubernetes. The MongoDB installation version is 7.0.1 and spans 62 machines, with a replication level of 3. The Druid database version is 29.0.1, and spans 24 machines, with 3 brokers, overlords, routers, and coordinators and 24 historicals and middle managers processes. The PostGres database is a single containerized instance. We pulled the container image from Dockerhub and used tag version 10.1.

B. *Datasets*

We validate our methodology using three publicly available datasets stored in diverse backend data stores:

- Stored within our Druid database, is the MACA (Multivariate Adaptive Constructed Analogs) Future Climate Dataset which is based on the RCP8.5 climate projections using the CMIP5 model. The size of this dataset is just under 1TB with over 8 billion observations, spanning from 2026-2075 [32].
- Within our PostGres database we use census data categorizing uncertainty and the magnitude of error for the 2020 U.S. Census Statistics using an approximate Monte Carlo Simulation. This dataset is 34 GB and has approximately 310 million observations [33].
- For our MongoDB database, we use a subset of the North American Mesoscale Forecast System (NAM) dataset provided by the National Oceanic and Atmospheric Administration (NOAA). This dataset uses a weather forecasting model to predict weather phenomena. The temporal range is 2010–2015, with a size of 25GB [34].

Datasets encompass the contiguous United States (CONUS).

C. *Query Latencies* [RQ-1, RQ-3]

In order to evaluate the efficacy of our methodology, we compare the latencies in a containerized and non-containerized environment for the set of queries which return data to the user. We performed 30 iterations of each query, calculated the standard deviations. The results from our benchmarks can be seen in Table II.

To create an accurate representation of latency for a user when using SCRYBE, where the queries are issued via a browser and then are routed from the client's container to a containerized cluster of servers. We created Python scripts that execute within a replicated user environment. This environment was containerized and deployed via Kubernetes. We then executed the same script in this replicated environment directly on a host machine which queried a single instance of a server running directly on a different host machine.

TABLE II
Average query latency and standard deviations for all data returning queries across all databases. We additionally evaluate containerized and non-containerized deployments to determine impact on latency.

Database	Query	Average Latency (Un-Containerized)	Standard Deviation (Un-Containerized)	Average Latency (Containerized)	Standard Deviation (Containerized)
MongoDB	Less Than	0.3925	0.3420	0.1153	0.0104
MongoDB	Greater Than	0.2754	0.0061	0.1167	0.0042
MongoDB	Equal To	0.9002	0.0383	0.3164	0.0067
MongoDB	Not Equal To	0.2489	0.0046	0.1135	0.0110
MongoDB	Random Sample	17.5265	2.2540	16.2910	1.4221
MongoDB	Get DataSet Features	0.0620	0.0018	0.0065	0.0004
MongoDB	Sample Categories	4.029804467	0.0022	3.148843708	0.0014
MongoDB	Sample Distribution	3.790938727	0.0281	1.615975145	0.0051
MongoDB	Sample Spatial Extents	16.78595133	0.0327	5.20677601	0.0053
Druid	Less Than	0.1732	0.2406	0.1418	0.2324
Druid	Greater Than	0.0434	0.0053	0.0835	0.1738
Druid	Equal To	6.9139	3.6215	10.9163	5.1993
Druid	Not Equal To	0.1144	0.2059	0.0588	0.1176
Druid	Random Sample	1.3898	4.0895	0.0355	0.0114
Druid	Get DataSet Features	0.0854	0.1319	0.0789	0.1708
PostGres	Less Than	0.0578	0.0665	0.2296	0.1599
PostGres	Greater Than	0.0329	0.0017	0.1492	0.0033
PostGres	Equal To	0.0330	0.0038	0.1553	0.0048
PostGres	Not Equal To	0.0327	0.0021	0.1530	0.0078
PostGres	Random Sample	143.7226	1.6940	143.9381	1.0431
PostGres	Get DataSet Features	0.0152	0.0041	0.0163	0.0012

We found that there was a 38.8% reduction in latency when the environment was containerized when querying our MongoDB installation. We believe this is due to the reduction of network communication between the server and the MongoS router instance. These MongoS routers act as the entry point into the distributed database, and as a result, all queries must be routed through a MongoS router regardless of where in the cluster it is located. In our server container instances, we co-locate within the Kubernetes pod a MongoS router to allow the container to directly query these routers without having to traverse the cluster to a separate host.

When comparing environments for our Druid database, we found a 22.93% latency increase when the client and server were containerized. We believe this is because there are additional networking costs associated to running applications within containers and there are no optimizations for accessing a Druid database when it is containerized, unlike MongoDB.

For our PostGres database, we found a 0.52% decrease in latency when the servers and clients were containerized. We believe this result is because our PostGres database is also deployed in a containerized environment, rather than running directly on hosts across a set of machines, like Druid and MongoDB. This database configuration ensures that their will be additional networking costs incurred because of its containerization regardless of the environment of the server that is issuing queries to it.

While query latency increases for one case (Druid) in our containerized architecture, the benefits of fault tolerance, workload scaling, automatic process restarts and scheduling, make using a containerized deployment advantageous for our methodology.

D. *Preservation of Resource Thresholds* [RQ-3, RQ-4]

*1) **Endogenous Mechanisms**:* For our endogenous mechanisms, rate throttling and sample size limiting, we performed experiments within a replicated user environment to determine if the mechanisms functioned. This environment was a client container accessed via a web browser hosted via JupyterHub. We restarted the environment after every iteration of the experiment to accurately capture the impact of the endogenous mechanisms.

To ascertain the effects of rate throttling on the user, we perform random sampling data requests at 1 second intervals and plot the sample size of the data returned by the server for each QoS. Requests are made continuously for 100 seconds, then a request cool-down period where the client remains idle occurs for 100 seconds. We then begin sending requests continuously at 200 seconds to determine if client sample sizes increase after the RPM statistics decreases. The resulting sample sizes were recorded and are shown in Fig. 6.

To demonstrate the impact of our sample size decay, we removed the request rate throttling mechanism to specifically illustrate the standalone effects of sample size decay and our active memory capping mechanism. For this experiment, data is requested at 1 second intervals for 1000 seconds for each QoS. The queried data is stored in a continuously growing pandas dataframe. As depicted in Fig. 7, we can see there is a decrease in sample size over time, followed by an immediate drop to 0 when the user reaches their maximum memory usage within their environment.

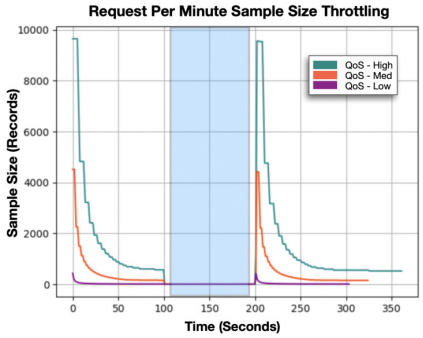

Fig. 6. This figure demonstrates our request rate sample size throttling. Between the times 100 sec and 200 sec we have a cool-down period, where the client remains idle. By doing this, the throttling is removed until they begin sending requests at time 200 sec. This additionally illustrates the impacts of QoS on initial sample size as well as rates of sample size decrease. We regulate our sample sizes based on the equations outlined in (1) and (2); the graph above demonstrates that the sample sizes adhere to these constraints.

We additionally illustrate the maximum memory allocation per client container in Fig. 8. Our methodology specifically prevents the user, regardless of QoS, from querying additional data from our system when it could exceed a specified endogenous memory cap. We have set this memory cap to be 80% of total memory allocated to a client container. We chose this limit to allow the user to still have resources available to perform analytics and explorations with the data they have queried.

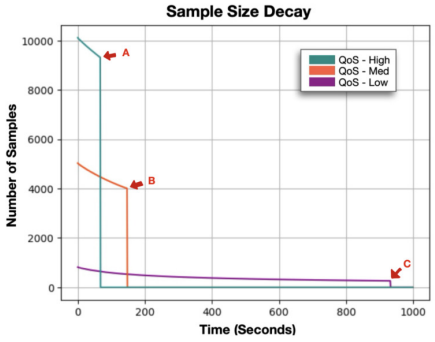

Fig. 7. This benchmark demonstrates the efficacy of the sample size decay and memory capping functionality in our methodology. SCRYBE controls the dataset sample size that a user is able to query and prevents the user from querying additional data if the user is at their maximum memory usage threshold. These thresholds are depicted at points A, B, and C.

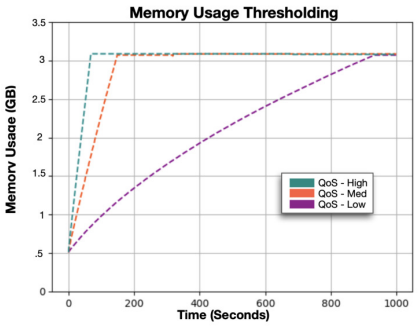

Fig. 8. SCRYBE regulates resource usage across memory and CPU. A consequence of this regulation is a reduction in the amount of data that a user is able acquire in order to prevent breaching resource thresholds. Regardless of the user's QoS, they are unable to query data if their active memory usage is above 80%.

Fig. 9. Efficacy of our methodology to prevent memory over consumption by a set of resource intensive containerized client processes over a 12 hr period. The red line shows the cumulative total memory allocation for the processes. This threshold is approached, but never breached.

*2) **Exogenous Mechanisms**:* To validate the effects of our exogenous thresholding mechanism, we have created a sandboxed environment to accurately discern the impacts of reg-

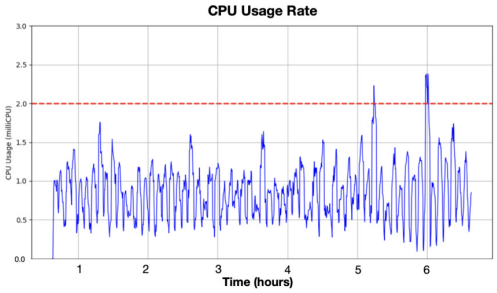

Fig. 10. SCRYBE performs CPU thresholding to prevent runaway client processes from over consuming cluster resources. CPU usage temporary breaches, but the measures reduce load to immediately return usage below threshold.

ulating resource over consumption. We have created a standalone Kubernetes Namespace and launched 3 server containers and 5 client containers. We configured the Namespace such that the cumulative total of CPU and memory usage cannot pass certain thresholds. These thresholds were set at 8GB and 2 milliCPU, which is equivalent to 2/1000 of a CPU. Upon launch, the client containers execute a Python script which mimics an extreme use case where a user continuously sends requests to the servers and stores the responses received into a Pandas Dataframe [28]. This script was specifically designed to be resource intensive.

To gather aggregated memory usage and CPU statistics, we configured Prometheus to monitor the Kubernetes Namespace such that we could accurately measure these statistics across the disparate containers. Prometheus is a framework that gathers and stores time-series system metrics across a network and can be additionally configured to interface with the Kubernetes API [35].

Based on the data we gathered from Prometheus, we can conclude the exogenous thresholding mechanism **effectively prevents the individual client containers from exceeding their memory allocation** by restarting the container when they breach 400MB of memory usage. The cumulative total of all containers within the Namespace is also prevented from reaching 8GB of memory usage. This behavior is illustrated in Fig. 9. This particular figure was generated from a 12 hr long experiment.

By analyzing the Prometheus data, we also determined that the CPU threshold temporarily breaches thresholds but the system quickly regulates itself to reduce CPU usage by restarting containers. This behavior is visible in Fig. 10.

*3) **Combined**:* We used the same experimental setup as with exogenous thresholding; however, the client container's requests additionally contain container memory usage and RPM statistics, which is then used by the server to determine how much data the client may receive. To create a more representative scenario of client behavior, we added a cool-down period after the client container sends requests continuously for 100 seconds. This cool-down period resets the client container's RPM statistics.

We compared the resulting memory usage with the exogenous only thresholding solution and found **a 11.67% reduction of memory** usage over a 12 hr period. This equates to **a total of 746.93 GB less total memory used**. When comparing CPU usage over a 6 hr period, **we found a 12.81% reduction, totaling 84.6 milliCPUs**. This results shows the efficacy of SCRYBE'S thresholding mechanisms when used in conjunction to prevent resource over consumption beyond SCRYBE'S allocation of resources.

TABLE III
SCRYBE provides built-in, interactive visualizations. Where possible, we leverage the client's GPU to reduce rendering times and alleviate server load.

Visualization	Time to Render (ms)	Client GPU Utilization (ms)
Chart Data Distribution	2152	9
Scatter Plot	1040	13
Violin Plot	1116	10
Box Plot	1119	16
ARIMA	8986	67
SARIMA	5674	51
KDE	1659	31
Choropleth	4310	106
Random Forest	2371	24

E. Client-side Visualizations [RQ-2]

To determine the efficacy and usability of our visualization techniques, we profiled our system using a qualitative approach that focuses on the quality of the user interaction with the SCRYBE'S browser-based interface. To achieve this, we performed several user experience evaluations using Google's Lighthouse browser auditing tool. This tool measures several aspects of a sites performance, but we chose to use rendering time and GPU utilization to perform our evaluation. To perform these audits, Lighthouse mimics an average computer with a slow internet connection and measures the responsiveness and interactivity of the site [36].

To ascertain the interactivity and usability of our brower-based, Jupyter Notebook visualizations, we compare our metrics collected from Lighthouse against the RAIL model [37]. This model was developed by Google Chrome to provide guidelines for how long certain web application interactions should take to remain interactive from the user's perspective. For this evaluation, we performed 10 audits for each visualization technique and took the average. The results of these audits are displayed in Table III.

In accordance with the RAIL model's interactivity standards for load duration until time to interact, this type of visualization needs to occur within 5 seconds from when the user issues the request. We found that all of our visualizations that did not require model training (SARIMA & ARIMA) performed within this time frame and as a result are interactive from a user-centric perspective.

V. CONCLUSIONS & FUTURE WORK

In this study, we have described our methodology for supporting programmatic explorations of voluminous datasets.

RQ-1: Providing simplified, extensible interfaces to data accesses allows us to reconcile complexity of diverse, distributed storage systems. This allows us to leverage diverse storage frameworks with possible extensions to other types of storage systems.

RQ-2: Leveraging sampling and client-side GPU accelerations allows us to preserve interactivity during visualizations. Crucially, none of the visual elements need to be computed server-side. Leveraging streaming also allows us to preserve interactivity by enabling incremental data transfers.

RQ-3: Leveraging containers, in particular, support for namespaces and configuration of resource thresholds allows us to ensure that server-side resources aren't overwhelmed. Because our queries return a representative sample of the data rather than the entire collection of records that satisfy the request, it conserves both disk I/O and network I/O on the server side.

RQ-4: Our differentiated services facilitate role-based resource configuration thresholds and sample sizes. We supplement this with constant monitoring of resource utilizations to dynamically increase thresholds when there is slack in resource utilizations. Individually and collectively the notebooks are not allowed to breach resource usage thresholds configured server-side.

For future work, we plan to explore supporting additional compute intensive operators such as principal component analysis and model fitting operations based on gradient boosting. A further avenue is to extend these programmatic interfaces to Scala and explore interactions with the Spark data analytics ecosystem.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation (1931363, 2312319), the National Institute of Food Agriculture (COL014021223), and an NSF/NIFA Artificial Intelligence Institutes AI-CLIMATE Award [2023-03616].

REFERENCES

[1] I. Salesforce, "Tableau," accessed: 2024-09-18. [Online]. Available: https://www.tableau.com/
[2] Q. D. Team, "Qgis." [Online]. Available: https://www.qgis.org/
[3] M. P. Papazoglou and W.-J. V. D. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, vol. 16, pp. 389–415, 2007.
[4] N. Serrano, J. Hernantes, and G. Gallardo, "Service-oriented architecture and legacy systems," *IEEE Software*, vol. 31, no. 5, pp. 15–19, 2014.
[5] J. Farrell and H. Lausen, "Semantic annotations for wsdl and xml schema," W3C Recommendation, 2007.
[6] C. Pautasso and E. Wilde, "Restful web services: principles, patterns, emerging technologies," in *Proceedings of the 19th International Conference on World Wide Web*, 2010, pp. 1359–1360.
[7] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, 2009.
[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, 2008.
[9] S. Garfinkel, "An evaluation of amazon's grid computing services: Ec2, s3 and sqs," August 2007.
[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the European Conference on Computer Systems*, 2007, pp. 59–72.
[11] J.-P. Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder, "An enabling framework for master-worker applications on the computational grid," in *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, 2000, pp. 63–70.
[12] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, Feb–April 2005.
[13] G. v. Laszewski and M. Hategan, "Workflow concepts of the java cog kit," *Journal of Grid Computing*, vol. 3, no. 3-4, pp. 239–258, 2005.
[14] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. v. Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *Proceedings of the IEEE Workshop on Scientific Workflows*, 2007, pp. 199–206.
[15] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *Proceedings of the 14th IEEE International Conference on Scientific and Statistical Database Management*, 2002, pp. 37–46.
[16] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, 1999, pp. 45–58.
[17] J. A. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace, "The spring system: Integrated support for complex real-time systems," *Real-time Systems*, vol. 16, no. 2/3, pp. 97–125, 1999.
[18] G. Singh, C. Kesselman, and E. Deelman, "Performance impact of resource provisioning on workflows," Information Sciences Institute, Tech. Rep., 2006, iSI.
[19] G. Mehta, C. Kesselman, and E. Deelman, "Condor-g: A computation management agent for multi-institutional grids," Information Sciences Institute, Tech. Rep., 2006, iSI Tech Report.
[20] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*. IEEE, 2012, pp. 73–80.
[21] Lloyd, Wes and Pallickara, Shrideep and David, Olaf and Lyon, Jim and Arabi, Mazdak and Rojas, Ken, "Migration of multi-tier applications to infrastructure-as-a-service clouds: An investigation using kernel-based virtual machines," in *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, 2011, pp. 137–144.
[22] G. Fox, S. Pallickara, M. Pierce, and H. Gadgil, "Building messaging substrates for web and grid applications," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 363, no. 1833, pp. 1757–1773, 2005.
[23] Y. Yan, Y. Huang, G. C. Fox, S. Pallickara, M. E. Pierce, A. Kaplan, and A. E. Topcu, "Implementing a prototype of the security framework for distributed brokering systems." in *Security and Management*, 2003, pp. 212–218.
[24] I. Docker, "Docker," https://www.docker.com/, 2023, accessed on 03/23/2023.
[25] JupyterHub, "Jupyterhub development," 2024, accessed: 2024-11-09. [Online]. Available: https://jupyterhub.readthedocs.io/en/stable/
[26] H. Wang and M. Song, "Ckmeans.1d.dp," *The R Journal*, vol. 3, no. 2, pp. 29–33, 2011.
[27] S. Overflow, "Stack overflow developer survey 2023," 2023, accessed: 2024-11-09. [Online]. Available: https://survey.stackoverflow.co/2023/
[28] Pandas, "pandas.dataframe," 2024, (Accessed: 2024-08-28). [Online]. Available: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html
[29] Google. (2024) grpc: Google remote procedure call. Accessed: 2024-08-29. [Online]. Available: https://grpc.io/
[30] Kubernetes. (2024) Kubernetes: Container orchestration. Accessed: 2024-08-29. [Online]. Available: https://kubernetes.io/
[31] Deck.gl, "Pydeck," https://deckgl.readthedocs.io/en/latest/, 2024.
[32] Climatology Lab. (2024) Maca. Accessed: 2024-09-05. [Online]. Available: https://www.climatologylab.org/maca.html
[33] USCB. Estimating confidence intervals for 2020 census. Accessed: 2024-09-05. [Online]. Available: https://registry.opendata.aws/census-2010-amc-mdf-replicates
[34] R. S. Vose *et al.*, "Improved historical temperature and precipitation data," *Journal of Applied Meteorology and Climatology*, vol. 53, no. 5, pp. 1232–1251, May 2014.
[35] Prometheus, "Prometheus," 2023, accessed: 2024-09-19. [Online]. Available: https://prometheus.io/
[36] G. LLC, "Lighthouse auditing tool," https://github.com/GoogleChrome/lighthouse, 2023.
[37] Google, "Rail: Measure performance," March 2023, accessed on 2024-08-28. [Online]. Available: https://web.dev/rail/