Trident: Distributed Storage, Analysis, and Exploration of Multidimensional Phenomena

Matthew Malensek, Walid Budgaga, Ryan Stern,
Shrideep Pallickara, and Sangmi Lee Pallickara, *Members, IEEE*

Abstract—Rising storage and computational capacities have led to the accumulation of voluminous datasets. These datasets contain insights that describe natural phenomena, usage patterns, trends, and other aspects of complex, real-world systems. Statistical and machine learning models are often employed to identify these patterns or attributes of interest. However, a wide array of potentially relevant models and parameterizations exist, and may provide the best performance only after preprocessing steps have been carried out. Our distributed analytics platform, TRIDENT, facilitates the modeling process by providing high-level data exploration functionality as well as guidance for creation of effective models. TRIDENT handles (1) data partitioning and storage, (2) metadata extraction and indexing, and (3) selective retrievals or transformations to prepare and generate training data. In this study, we evaluate TRIDENT in the context of a 1.1 petabyte epidemiology dataset generated by a disease spread simulation; such datasets are often used in planning for national-scale outbreaks in animal populations.

Index Terms—Distributed analytics; voluminous data management; machine learning

1 INTRODUCTION

RECENT advancements in distributed storage and computation engines have enabled analytics at an unprecedented scale, with systems such as Spark [1] and Hadoop [2] allowing users to build distributed applications to gain insight from voluminous, multidimensional datasets. While these systems are highly effective from a computational standpoint, both exploration and *feature engineering* for machine learning models require several rounds of computation and incur I/O costs as data is migrated into main memory.

To address these use cases we propose TRIDENT, which supports three key aspects of handling data in the context of analytic modeling: (1) distribution and storage, (2) feature space management, and (3) support for ad hoc retrieval and exploration of model training data. Incoming feature vectors are partitioned to facilitate targeted analysis over specific subsets of the feature space. Transformations supported by TRIDENT include normalization, binning, and support for dimensionality reduction based on correlation analysis. Exploration and retrieval of model training data is enabled by expressive queries that can prune the feature space, sample across feature vectors, or combine portions of the data. Exposing this functionality at the storage level (rather than in a computation engine) allows many steps in the feature engineering process to be performed before analysis begins. By leveraging this functionality, researchers and practitioners can explore and inspect their datasets in an interactive fashion to help guide the creation of machine learning models or visualizations without needing to write ad-hoc applications or wait for heavyweight distributed computations to execute.

1.1 Scenario

Consider a TRIDENT user, or *analyst*, that wishes to explore a voluminous, high-dimensional epidemiology dataset stored in the system. The dataset is likely to contain valuable insight into a phenomenon that occurs when a particular series of events unfold, and the analyst wishes to understand what factors are most influential by creating analytical models of the data. For example, such analysis could involve locating feature vectors that occurred when the disease outbreak duration was negatively correlated with vaccine stockpiles. The analyst begins by issuing a set of SQL-like queries in an interactive console to pinpoint occurrences of the phenomenon, and then requests a statistical summary of the matching feature vectors to understand their distributions. At this point, additional constraints may be placed on the scope of the inquiry: perhaps the analyst finds that winter months have a stronger impact on vaccine stockpiles due to reduced shipping capacity, and decides to focus on a particular temporal scope in order to be more selective.

During the exploration process, the analyst can adjust query parameters interactively and receive real-time feedback; feature information is retrieved and processed on the server side to avoid transmitting large quantities of data to the analyst's workstation. On the client side, metadata in the form of *analytic base trees* (ABTs) compactly represent the dataset, its relationships, and statistical information. Once the analyst is finished selecting a subset of the feature space, he/she locates atypical scenarios by retrieving data points where the disease duration was more than a standard deviation away from the mean, and prunes features that do not correlate with the disease duration. Satisfied with the dataset, the analyst normalizes the feature values by scaling them into a range of [0, 1], trains exploratory machine learning models in parallel, and subsequently retrieves a *bias-variance decomposition* report. This report helps the analyst understand any forecasting errors and guides further modeling efforts; as new data is streamed into TRIDENT, the analyst may also revisit previous analyses or reconfigure query parameters.

1.2 Research Questions

Supporting storage and analytics activities in this domain introduces unique challenges due to high dimensionality, dataset sizes, and the latency requirements during interactive queries. We developed the following research questions to guide our implementation of TRIDENT:

RQ1 How can we control the placement of incoming feature vectors to improve retrieval times and construction of specialized models? [§3]

RQ2 How can we efficiently support analytic operations on specific portions of the feature space while avoiding disk I/O? [§4]

RQ3 How can we provide a rich set of queries that facilitate analytics and minimize duplicate processing (or preprocessing)? [§4, §5]

RQ4 How can we facilitate creation and evaluation of models for particular portions of the feature space? [§5]

1.3 Approach Summary

TRIDENT is a holistic, integrated framework that targets both *how* and *where* training data is stored in the system. Data partitioning can be configured using multiple strategies, including hash-based and spatially-aware partitioners. The default partitioner performs correlation analysis between independent and dependent variables to achieve dimensionality reduction. Reduced-dimensionality feature vectors are then clustered and dispersed to storage nodes that hold similar data. Clustering data points with high similarity enables the creation of specialized models that outperform models generated with randomly-placed data.

The system extracts metadata from incoming feature vectors to populate our novel *analytic base tree* (ABT) indexing structure. ABTs track relationships between features and are used to evaluate distributed queries. To handle dimensionality, ABTs support autonomous quantization to control memory consumption and ensure low-latency retrievals. During the feature space management process, we also support preprocessing operations such as additional dimensionality reduction steps, feature ranking, and normalization.

Our distributed query support allows the analyst to explore the feature space and identify training data points of interest with an SQL-like interface. Standard relational operations are supported, as well as analytic operators such as event frequencies, sampling, and *fuzzy* queries that allow constraints to be relaxed. After selecting relevant feature vectors, analysts can perform an automated *bias-variance* decomposition of pilot models and export in-memory datasets for further manipulation and training by outside frameworks. We demonstrate this capability by generating models with Spark [1], TensorFlow [3], and a custom machine learning application based on scikit-learn [4].

1.4 Contributions

TRIDENT is focused on storage-level innovations to improve the overall responsiveness of modern analytics pipelines. The system actively inspects and analyzes multidimensional data points as they are being stored; in contrast, distributed file systems like HDFS [5] are more general but do not provide query functionality or produce metadata based on incoming dimensions and their relationships. Such metadata is vital when providing a near-instantaneous, real-time feedback loop for analysts. When it is efficient to do so, we also provide built-in preprocessing and modeling facilities. Specific contributions of TRIDENT include:

- A fast, query-driven approach to feature space exploration and model construction, with a rich set of queries that support retrieval of training data.
- Modeling guidance provided by automated dimensionality reduction, bias-variance decomposition, and ad hoc creation of pilot models.
- Training data management for fast, targeted retrievals that can be exported to formats including Spark RDDs [6], LIBSVM [7], and TensorFlow [3] records.

TRIDENT does not replace existing computational frameworks such as Spark or Hadoop; rather, it integrates into the existing ecosystems and provides augmented exploration and transformation capabilities that do not require latency-prone disk I/O to occur across the entire dataset during each operation.

2 TRIDENT: METHODOLOGY

The crux of our effort is to help analysts construct relevant analytical models. A key theme underpinning these core capabilities is the preservation of timeliness, allowing the analyst to quickly identify interesting data, gather insights, fit models, and assess their quality.

To contrast with other approaches, consider a basic computational operation — retrieving the average (mean) of a particular feature. While straightforward in an algorithmic sense, this requires heavy disk and memory I/O in systems such as Hadoop or Spark, whereas in TRIDENT the operation can be completed in less than **1 ms** by querying our indexing structure. Since the metadata collected by the system is general and can be fused, filtering such a query based on time or additional feature values does not incur additional latency.

TRIDENT is designed to assimilate data incrementally as it arrives, allowing both streaming and in-place datasets to be managed. The system employs a network design based on distributed hash tables (DHTs) to ensure scalability as new nodes are added to its resource pool, and uses a gossip protocol to keep nodes informed of the collective system state. This allows flexible preprocessing and creation of training data for statistical and machine learning models. Our methodology encompasses three core capabilities:

1) ***Data Dispersion*** (§3): Effective dispersion of the dataset over a collection of nodes underpins data locality, representativeness of in-memory data structures, and the efficiency of query evaluations. The resulting data locality promotes timeliness during construction of specialized models for different portions of the feature space.
2) ***Feature Space Management*** (§4): TRIDENT maintains memory-resident metadata to help locate portions of the dataset, summarize its attributes, and preprocess feature vectors. Online sketches ensure the data can be represented compactly and with high accuracy, while preprocessing activities enable operations such as dimensionality reduction or normalization.

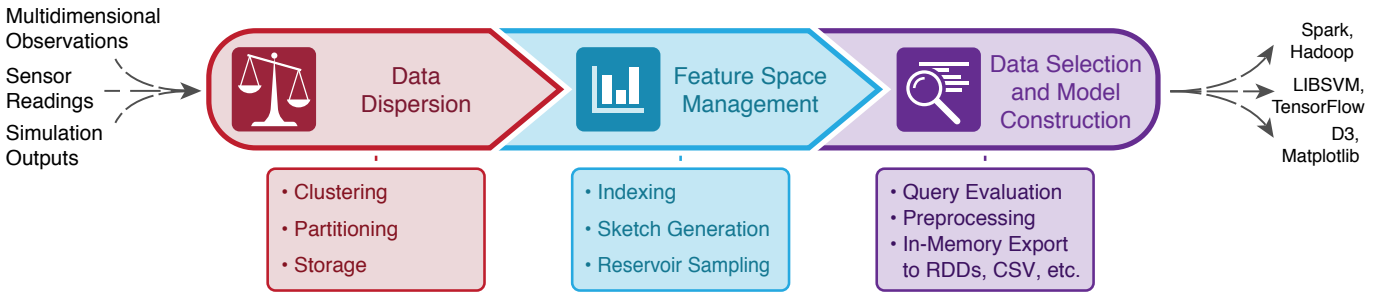


Fig. 1. TRIDENT architecture: multidimensional records are partitioned and indexed for subsequent analysis through expressive queries. Once records of interest are located and transformed, they are exported in memory to a variety of computation engines' native data structures.

3) ***Data Selection and Model Construction*** (§5): TRIDENT supports interactive exploration via *steering* and *calibration* queries to probe the feature space. These real-time queries help analysts sift and identify training data of interest. Training data can be exported to a variety of formats, including DataFrame implementations supported by R [8], Pandas [9], and Spark [1]. TRIDENT also manages training and assessment of analytical models via generation of cross-validation folds and bias-variance decomposition of model errors.

An overview of this functionality is shown in Figure 1. While we evaluate TRIDENT in the context of two representative datasets, our methodology does not preclude the use of data from other domains with similar dimensionality (hundreds to thousands of dimensions) where there is a need to understand phenomena or forecast outcomes.

2.1 Datasets: Epidemiological and Atmospheric

Epidemiological Dataset: Most examples throughout the text are based on our subject dataset, which was produced by a discrete event simulation, the *Animal Disease Spread Model* (ADSM) [10]. ADSM simulates disease outbreaks in livestock populations and has been used in several studies including foot-and-mouth disease, avian influenza, and pseudorabies. ADSM and its predecessor, the *North American Animal Disease Spread Model* (NAADSM) are Monte Carlo models, which means that each set of parameters, called a *scenario*, is run several times to gain confidence in the outputs. Running multiple iterations of ADSM is computationally expensive and can consume hours of CPU time. This makes exhaustively exploring the simulation parameter space untenable, and is a driving factor behind leveraging analytic models to gain high-level insights from the data.

In this study, we used an ADSM scenario set in Texas, USA, that simulated an outbreak of foot-and-mouth disease across 364,000 farms with direct contact, indirect contact, and airborne disease spread. The outbreak was based on real-world farm and disease biology data, with each scenario variant executed 30 times to account for uncertainty in the results. The overall dataset size was 1.1 petabytes, and included over 2000 unique features.

Atmospheric Dataset: To help demonstrate the flexibility of TRIDENT, we also used an atmospheric dataset collected from the NOAA NAM Forecast System [11]. The NAM contains atmospheric observations with a variety of features such as the date, time, location, surface temperature, humidity, wind speed, and precipitation. This dataset has fewer features (around 100), but more observations (20 billion files, 500 TB of data).

2.2 Test Environment

Benchmarks described in the following sections were conducted on a cluster of 75 nodes: 45 HP DL160 servers (Xeon E5620, 12 GB RAM), and 30 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM) running Fedora Linux version 26 (kernel 4.14). For data structure benchmarks carried out on a single node, the DL320e configuration was used. Each host was equipped with four hard disk drives, and TRIDENT was executed on the OpenJDK Java runtime, version 1.8.0_151.

3 DATA DISPERSION

TRIDENT is based on a distributed hash table (DHT) network design [12], [13], [14]. A DHT is a type of decentralized overlay network that is created by dividing a *hash space* across participating nodes, which ensures that incoming or outgoing hosts have a minimal impact on the system as a whole. This facilitates scalability in production settings. The DHT used by TRIDENT is constructed in a manner similar to Cassandra [15] or Amazon Dynamo [16], where requests are routed directly to their final destination rather taking intermediate hops through the network. This approach helps ensure predictable latencies and is beneficial during outages or failures, as the loss of a single node does not impact routing for the rest of the system. Storage and retrieval operations can be handled by any of the participating storage nodes, which will route the requests to their appropriate destination.

In the traditional *key-value* storage model employed by distributed hash tables, load is balanced in a roughly uniform fashion across participating nodes. This helps avoid *hotspots* in the network that receive a larger share of storage requests. When desired record *keys* (such as a filename or identifier) are known ahead of time, locating data is straightforward. However, searching for a particular *value* in a DHT instead of a predetermined key generally requires an exhaustive search to be broadcast across all participating nodes. Furthermore, the randomized placement of data across the cluster makes reducing the search space of a lookup operation impractical, hindering query capabilities. To address these concerns, TRIDENT supports multiple partitioning strategies that work in tandem with its

indexing functionality. These include the *uniform*, *geospatial*, and *cluster-based* partitioners, which can be extended or replaced with custom partitioner implementations. While the cluster-based partitioner is the default, users may select a different partitioner at run time based on their dataset or computations.

3.1 Uniform Partitioning

Most DHT-based storage systems, including TRIDENT, implement uniform partitioning via hash functions such as MD5, SHA-1, etc. With this strategy, each file has an associated *key* (often a file name or unique identifier) that is mapped to the overall hash space. Each computing resource is assigned a portion of the hash space, leading to a relatively uniform distribution of load; in our test environment, the uniform partitioner assigns each of the 75 nodes about **1.33%** of the load with a standard deviation of **0.08%** for both of our test datasets. The hash key can also be used to retrieve files directly, providing efficient lookup functionality. In situations where nodes in the cluster are heterogeneous or have different capabilities, machines may represent additional *virtual* nodes to take on additional load. While the uniform partitioner provides a reasonable baseline for all file types with uniform load balancing, selecting a data-specific partitioner in TRIDENT such as the geospatial or cluster-based partitioner can improve query latencies and the organization of records in the system.

3.2 Geospatial Partitioning

For lookup operations that prominently feature geospatial characteristics, balancing load across the TRIDENT cluster in a spatially-aware fashion can dramatically improve query resolution times due to collocation of spatially proximate records. We employ the Geohash [17] algorithm to generate a spatially-constrained hash space that identifies locations on the Earth as Base 32 strings. Geohash generates a hierarchy of bounding boxes where similar hash prefixes represent similar spatial locations; Geohashes *9XJ63* and *9XJ66* are both located in Denver, Colorado, USA, while *9QCE7* represents a region in Sacramento, California, USA. To handle varying spatial densities, the granularity of the hash space allocated by our Geospatial partitioner can be configured by specifying a Geohash string length; short strings allocate larger spatial regions to single nodes, while longer strings will distribute records based on finer-grained bounding boxes. While partitioning load based on spatial characteristics can greatly improve collocation for particular use cases, TRIDENT also allows multiple features to be considered for collocation via its cluster-based partitioner.

3.3 Cluster-Based Partitioning

To automate data placement, retain the beneficial load balancing characteristics of DHTs, and also facilitate search operations, we developed a *locality-sensitive hash* (LSH) function based on data clustering — one of the key contributions of the TRIDENT framework. By clustering incoming data points, we can place similar feature vectors on the same nodes while ensuring the number of clusters created is reflected by the number of storage nodes participating in the system. While cryptographic hash functions attempt to avoid *collisions* where multiple entities are mapped to the same hash, locality-sensitive hashing encourages such collisions. This results in feature vectors with high similarity being placed on the same storage node or neighboring nodes within the hash space.

In the initialization phase of our cluster-based partitioning algorithm, data points are buffered at each storage node and funneled to a *cluster manager* that is elected and gossiped through the system. As data arrives, the cluster manager begins populating a StreamKM++ clustering model [18] provided by the Massive Online Analysis (MOA) framework [19]. StreamKM++ is a streaming variant of k-means that uses the k-means++ algorithm to determine initial cluster *centroids* based on data distributions rather than using random locations. This improves cluster quality and the convergence speed of the algorithm. The initialization process ends when either (1) the algorithm converges on a set of clusters, or (2) the percentage of memory consumed by buffered data at any node reaches 25%. Once initialization is complete, the storage nodes are assigned a set of cluster centroids to manage.

During normal operation of the TRIDENT cluster, changes to the centroids are published by individual storage nodes on a periodic basis. This ensures routing decisions will respect the evolution of the feature space as new data is added to the system. Our gossip scheme is *eventually consistent*, meaning some feature vectors may be stored based on outdated information. However, our aim is not to produce an exact clustering but rather collocate data with reasonable precision, which in turn improves the performance of model creation and analytics activities.

The initial number of clusters (k) is set to the twice the number of storage nodes present in the system. As new nodes are added to the storage pool, they assume responsibility for these extra clusters. Cluster assignments are determined by retrieval and storage loads; nodes managing the highest amount of load are selected for migration first. This allows the cluster to scale out rapidly, but limits the maximum number of nodes that can be added to the system at a given time. To ensure further elasticity and handle situations where clusters become imbalanced, the system can *split* or *merge* clusters as well. These operations require additional coordination by the cluster manager to reassess centroids and potentially migrate data, which makes them best suited for manual initialization by a system administrator during periods of low traffic. Split and merge operations are generally localized to a small subset of the storage nodes, which can be computed before committing the changes.

3.4 Improving Cluster Quality

In situations involving high dimensionality, algorithms that represent data in Euclidean space (such as k-means) often suffer from high dispersion and sparsity [20]. As a result, clusters produced in such cases may be valid but not exhibit meaningful feature similarities. This is particularly problematic in the context of our partitioning scheme; without similar groups of data, value-based queries still require an exhaustive search across all the data in the cluster. To measure the quality of the clusters produced by our partitioning algorithm, we use the Davies-Bouldin (DB) index

[21]. The Davies-Bouldin index evaluates clusters based on their intra- and inter-cluster similarity; if the clusters feature low dispersion and do not overlap, they are more likely to represent distinct concepts, and therefore have a lower DB index. In general, minimizing the Davies-Bouldin index results in higher-quality clusters. The DB index is calculated as follows:

$$DB = \frac{1}{n}\sum_{i=1}^{n}\max_{j\neq i}\left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right)$$

Where n is the number of clusters, σ represents the average distance of data points from their cluster centroid, and $d(c_i, c_j)$ is the distance between centroids. Ideally, each storage node in TRIDENT will be assigned a well-defined portion of the feature space to manage, but there is a trade-off between the number of clusters generated by our algorithm, data dimensionality, and cluster quality. Since the number of clusters depends on the system configuration, we target dimensionality reductions to increase cluster quality. Our test dataset includes over 2000 unique features, but we found that many are used infrequently or are directly influenced by other features; for instance, if airborne disease spread does not occur, several other features will be less prominent. To discover these interactions between features, we perform online correlation analysis using the Pearson product-moment correlation coefficient (PCC) on incoming scenario data (inputs and outputs) to rank feature importance. PCC measures linear dependence between variables, where a correlation coefficient of -1 represents a strong negative relationship, +1 is a strong positive relationship, and 0 indicates no correlation between the variables.

Input variables that exhibit strong correlations with output variables often improve model accuracy. On the other hand, correlations between **inputs** can lead to *collinearity*, which occurs when two or more input features can be predicted accurately from one another. Our dimensionality reduction process executes on the cluster manager, which begins by calculating the cross-feature correlation coefficients of the normalized input and output features. Collinear inputs are removed first, with the remaining features ranked by the absolute value of their correlation coefficients. The bottom 25% of these features are removed, clusters are generated, and the Davies-Bouldin index is calculated for the configuration. This process continues until the resulting change in cluster quality is less than 1.0. We also allow analysts to tune these thresholds or provide an ordered list of features to prioritize/de-prioritize during dimensionality reduction if greater control is necessary.

3.5 Cluster-Based Partitioning Evaluation

After dimensionality reduction is complete, TRIDENT begins using the clusters to partition data. Storing our epidemiological dataset on our configuration of 75 machines resulted in the distribution of load shown in Figure 2. While there are small variations in the percentage of overall data stored at each node, our algorithm does not introduce extreme load imbalances. Furthermore, the dimensionality reduction process reduced the Davies-Bouldin index of the clusters from **32.3** to **4.1**.

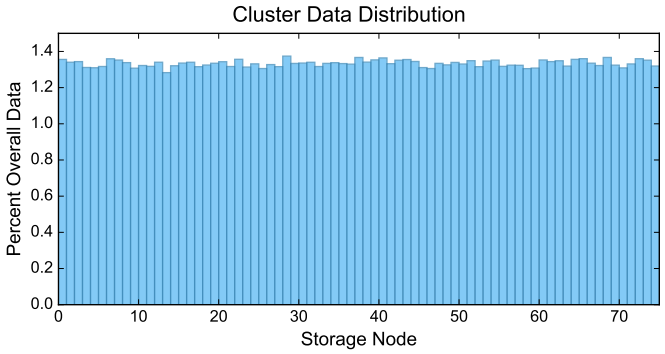


Fig. 2. Distribution of data with our cluster-based locality-sensitive hashing scheme across 75 storage nodes.

To further evaluate the efficacy of our partitioning strategy, we generated models using gradient-boosted decision trees [22], [23], [24] on each node in the system and then repeated the process with both HDFS as well as randomized placement used in a standard DHT. In this test, individual models were trained with the entire local dataset present at each storage node and were configured to predict the `disease_duration` output variable from our test dataset. Figure 3 demonstrates the difference in prediction accuracy between the clustered and randomized models, with an improvement of 5–9%. Note that the results are sorted in descending order by root-mean-square error (RMSE) of the models, and that in the case of HDFS there is a slight increase in error due to reduced uniformity in load balancing. By ensuring similar data points are placed on the same storage node we can help local models specialize for their particular portion of the dataset, resulting in better performance. This also improves models that sample across the TRIDENT cluster to build training data, as each node holds a distinct portion of the overall feature space. However, it is worth noting that the hash function in a standard DHT is used for both partitioning and retrieval capabilities, whereas our partitioning scheme does not provide key-based lookup functionality. To support queries, we build indexes that are used to locate and retrieve specific portions of the feature space during *feature space management*.

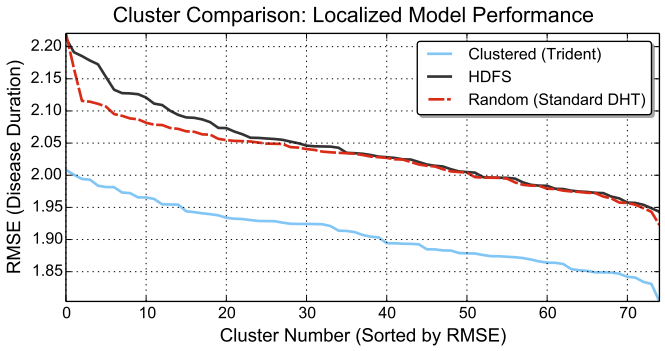


Fig. 3. Gradient-boosted decision tree models built in parallel with TRIDENT compared to models generated over randomly-placed data on a traditional DHT. Controlling the placement of data results in a 5–9% improvement in prediction accuracy.

4 Feature Space Management

With the dimensionality involved in our subject simulation and the problem domain in general, storing all feature vectors in memory or indexing every data point observed by the system is not feasible. To allow expressive query functionality, we employ streaming *sketches* to gather aggregate information about the data as well as tree-based, quantized indexes to provide lookup functionality. This approach gives analysts low-latency, iterative search capabilities while still enabling full-resolution retrievals from data stored on disk.

4.1 Online Sketches

TRIDENT supports compact, memory-resident sketches that provide a variety of metadata, including:

- Summary statistics (min, max, mean, etc.) for each feature
- Kernel density estimation (KDE) to provide probability densities for each feature
- Reservoir samples, which create online, representative sample sets of unbounded data streams

As multidimensional feature vectors stream into the system, *summary statistics* are a concise way to provide overviews of the data. Summaries include information such as minimum and maximum values, means, standard deviations, and variances. To provide this functionality without needing to inspect the entire dataset, we use Welford's Method [25]. Welford's Method allows TRIDENT to maintain summaries that are updated incrementally as data is stored, where each summary consumes about 40 bytes of memory. To expand the scope of these summaries, we also observe cross-feature relationships, which enable calculation of the Pearson product-moment correlation coefficient, coefficient of determination (r^2), and creation of two-dimensional linear regression models. This facilitates cross-variable analysis; for example, TRIDENT supports queries such as "what is the likely scenario infection rate when the radius of vaccination is three kilometers?" As discussed in the previous section, correlation coefficients provided by this functionality are also used by the system to help reduce dimensionality during data partitioning and storage.

Summary statistics maintained using Welford's method can also be *merged* to create aggregate statistics. We leverage this functionality to allow statistics to be gathered across different nodes in the TRIDENT cluster and then merged to form a single, coherent summary. This is particularly useful when a query involves several storage nodes; merges are a lightweight, streaming operation that can also be performed on the client side.

Another sketch supported by TRIDENT is online kernel density estimation (KDE). KDE provides an approximation of the probability density function (PDF) of a given variable based on sample data. We use the oKDE library [26] to maintain an online model of the estimated probability densities of each input and output variable in the system. This provides information on how each variable is distributed, what the most common values are, and how likely particular ranges of values will be. To account for evolution in the feature space, oKDE allows an optional *forgetting factor* to phase out old data over time, as well as a configurable *compression* to influence how much data is retained in memory. This allows dynamic management of the memory-accuracy trade-off space associated with each feature PDF. For our subject dataset, we found that evolution in the feature space did not occur rapidly enough to require old data to be phased out, but we do adjust the compression level based on memory constraints at the storage nodes.

To maintain full-resolution data points in memory, TRIDENT employs *reservoir sampling*, which enables a representative sample to be constructed and updated as new data points arrive [27]. Reservoir sizes are configured based on the available memory at each storage node. As new feature vectors stream into the system, they are randomly selected to be inserted into the reservoir with a decreasing probability. As a result, updates to the data structure become less frequent as more data is inserted, ensuring that the sample adequately represents the breadth of the dataset. Much like the other sketches implemented in TRIDENT, reservoir samples can be merged to form a single aggregate sample. Additionally, analysts that wish to build machine learning models across the entire dataset can quickly access the in-memory reservoirs to populate and train a model without requiring expensive disk I/O operations.

Compared to the other online sketches maintained by TRIDENT, reservoir samples consume a more significant portion of main memory — up to 25% of the total RAM by default. If distributed model generation (launched directly by TRIDENT or through a computation framework such as Hadoop [2] or Spark [1]) requires additional memory at the storage nodes, reservoir samples can be dropped by request and then subsequently restored after the tasks complete. This is facilitated by the system *reservoir journal*, which contains pointers to the full-resolution feature vectors stored on disk. The reservoir journal is updated as changes are made to the samples and logged to stable storage, ensuring the reservoirs can be removed from memory immediately if required.

While sampling is often an effective means to represent a dataset without inspecting every feature vector, we also use our online summary statistics to give analysts and client applications a sense of how accurate the reservoirs are. Table 1 demonstrates a *reservoir statistics report* generated by TRIDENT for the `disease_control_zone` input variable. Error percentages are included with the report to help analysts gauge the relative difference between the sample and actual values from the dataset. It is worth noting that random sampling captures high-level insights from the dataset, but may underrepresent outliers. To locate such data, evaluate queries, and analyze the feature space, we provide *analytic base trees*.

4.2 Analytic Base Trees

To provide query capabilities over the multidimensional data stored in TRIDENT, we developed a tree-based, hierarchical index for input and output features called *analytic base trees* (ABTs). Analytic base trees are sparse data structures that allow query operations to *drill down* through the feature hierarchy in a fashion similar to traditional file system interfaces. Each additional feature value or range specified by a query reduces the search space until a final set of matching feature vectors has been constructed. Besides enabling query evaluations over the files at a TRIDENT storage

TABLE 1
Reservoir statistics provided for a 1% sample of the `disease_control_zone` input variable.

Statistic	Actual Value	Sample Value	Error
Mean	22.06	22.15	0.41%
Variance	73.10	73.76	0.90%
Std. Dev.	8.55	8.59	0.47%
Min	4.61	4.73	2.60%
Max	35.40	35.37	-0.08%

node, ABTs also maintain online summary statistics for each *path* through the tree using our implementation of Welford's method. Unlike the general statistics maintained by the storage nodes for each feature type, the statistics stored in ABTs are merged dynamically to generate summaries for certain conditions or combinations of events.

Each storage node maintains an ABT instance for the input and output datasets under its purview, which are linked by feature vector instance identifiers. As information streams into the system, multidimensional feature vectors are converted into hierarchical paths that represent a traversal through the tree. Each feature value becomes a tree vertex, with edges used to connect related data points. Multiple layers in the hierarchy are used for interval data or types of features that include additional sub-dimensions. Leaf nodes contain file pointers to the locations of feature vectors on stable storage, as well as summary statistics for their corresponding paths. This approach not only maintains the multidimensional relationships between features, but also conserves memory by enabling *vertex reuse* for duplicate feature values. Figure 4 provides a simplified demonstration of the ABT; while the example contains six feature vertices and five leaves, production environments would involve millions of vertices, edges, and leaves.

ABT queries support a SQL-like fluent syntax that allows analysts to search by range, exact match, inequality, or substrings. During query evaluation, TRIDENT performs a traversal and creates a set of *pruned* vertices based on paths

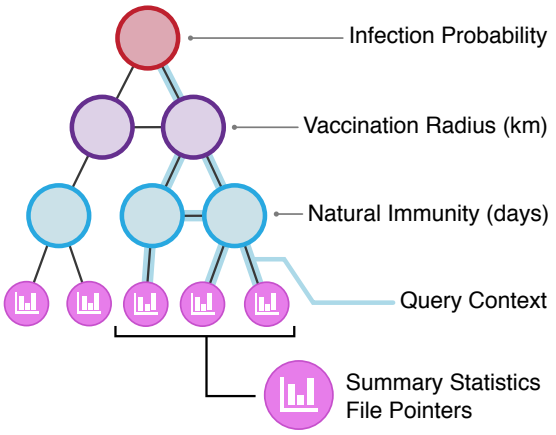


Fig. 4. An example analytic base tree with three feature layers. A query context, shown with highlighted edges, prunes the feature space to select particular sets of files or summary statistics.

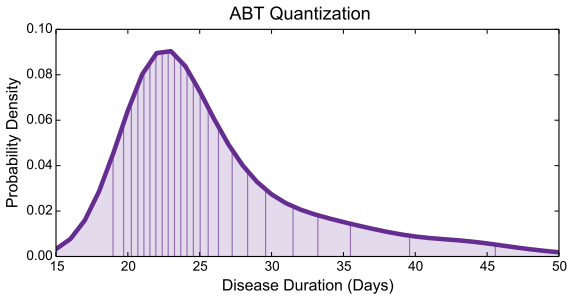


Fig. 5. A demonstration of the autonomous binning supported by analytic base trees. Feature values with higher probabilities are placed in smaller (and therefore more accurate) bins.

that do not match the query. This allows irrelevant portions of the tree to be eliminated without being traversed. Constructing the set of pruned vertices creates a limited *query context*, which can be queried further, serialized and streamed to clients, or used to inspect statistics that were stored under the matching paths. In many cases, the tree itself provides enough information about the feature space without requiring disk accesses. Figure 4 demonstrates a sample query context generated by a request that matches the *infection probability* and *vaccination radius* exactly, and implicitly includes two *natural immunity* vertices.

While vertex reuse helps limit the memory consumption of the ABTs, high dimensionality can produce long, sparse paths that rarely result in duplicate values being collapsed into a single vertex. TRIDENT leverages the dimensionality reduction performed during data partitioning to determine which features to index, ensuring any user-specified key features are included. To further increase vertex reuse, we employ *autonomous binning* to quantize incoming data points. Autonomous binning places feature values into small, range-based *bins*; for instance, a single vertex would be responsible for all infection probabilities in the range 0.10 to 0.15 instead of only a single value.

We dynamically generate bins by leveraging the online kernel density estimation functionality in TRIDENT. For each feature, corresponding probability densities are used to create range boundaries that distribute data points uniformly across the bins. Bins are generated during the cluster initialization phase, and updated periodically at run time if substantial changes in the distributions occur. Updates are performed on an incremental, targeted basis, where bins are either split in two or merged with a neighboring bin. Figure 5 illustrates this process for the *disease duration* output variable; note how ranges with the highest concentration of values are smaller, increasing precision, while outliers are placed in larger, more general bins. To further improve accuracy, binning is performed on a case-by-case basis at each storage node, reflecting the unique properties of each clustered portion of the data. Table 2 reports memory consumption and statistics for the analytic base tree before and after our autonomous binning process. Overall, the process reduced memory consumption by about 64% with our test dataset. This enables the storage nodes to manage more features with higher dimensionality, and also helps improve caching performance for frequently-used feature

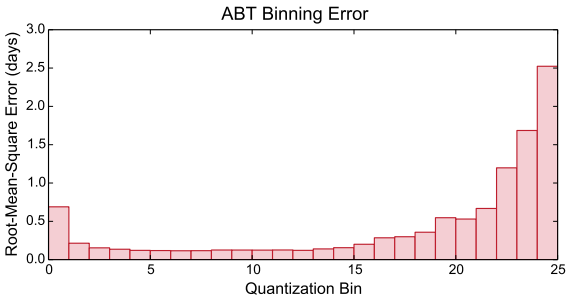


Fig. 6. Root-mean-square error for each range of values created by our autonomous binning algorithm. Frequently-observed values are placed in the smallest ranges, resulting in higher accuracy.

vectors during the iterative modeling process.

TABLE 2
Analytic base tree metrics before and after autonomous binning.

Metric	Original	Binned	Change
Vertices	11,363,106	3,827,937	-66.3%
Edges	12,414,085	4,321,440	-65.2%
Leaves	1,050,980	493,504	-53.0%
Memory	3,539.4 MB	1,268.5 MB	-64.2%

Autonomous binning can substantially reduce memory consumption. However, these improvements come at the cost of decreased accuracy in queries. To evaluate the difference in accuracy incurred by autonomous binning, we compared results returned by the ABT with the corresponding files stored on disk. Since the vertices represent a range of values, we used the mean provided by our statistics instances as the value reported by the tree. Figure 6 measures the root-mean-square error of the values produced by the ABT; for the majority of the bins, the reported values' RMSE was less than 0.25 simulation days. However, for rare events — in this case, a disease outbreak lasting over 45 days — the RMSE was about 2.5 days. We report these error bounds alongside query results; also note that when full-resolution files are retrieved they are evaluated against the query parameters to ensure accurate results.

Due to the constant evolution of the feature space, our cluster-based partitioning scheme cannot be used to locate relevant storage nodes for query routing. Cluster centroids are allowed to drift over time to account for changes in the underlying data stream, which introduces the potential for *false negatives* (nodes that are incorrectly flagged as not having relevant data). Additionally, use of the uniform or geospatial partitioners limit retrievals to a single dimension (hash keys and locations, respectively). To avoid broadcasting queries to all nodes in the system, which would incur needless latency and processing overheads, we provide a *global* ABT. The global ABT is structured similarly to its local counterpart, but leaf nodes contain matching storage node identifiers instead of file locations. Additionally, our autonomous binning procedure is configured to produce much coarser-grained bins in the global version of the tree to ensure its memory consumption will be low. The global ABT is updated periodically as changes occur in local storage nodes' indexes, with updates gossiped in an eventually-consistent fashion. On average, updates to the global ABT are less than 1 MB, and can be merged directly with existing instances. An important consideration for the global variant of the tree is that due to its hierarchical nature *false positives* are possible, but false negatives are not. In the case of a false positive during distributed query evaluation, nodes will use their local ABTs to quickly resolve these *null* queries.

4.3 Preprocessing Operations

The sketches and indexing structures leveraged by TRIDENT also enable a variety of preprocessing operations, including correlation analysis and normalization. These activities are often carried out before training machine learning models, and can lead to better model performance. Correlation analysis is provided by the Pearson product-moment correlation coefficient (PCC) available in our cross-feature summary statistics. TRIDENT also allows correlations to be sorted based on the strength of the inter-feature relationship, and can retrieve the top-k items to reduce dimensionality. Normalization support implements *feature scaling*, which uses summary statistics to retrieve the range of values for a particular feature and then scales each value (F') to fall within a range $[a, b]$:

$$F' = a + \frac{(X - X_{\min})\,(b - a)}{X_{\max} - X_{\min}}$$

This helps avoid issues where features with large values or extreme variation outweigh others in the dataset, which some machine learning and statistical models are less resilient to.

5 Data Selection and Model Construction

TRIDENT provides several types of queries to support iterative, ad hoc exploration of the input and output feature space. This includes relational queries on feature values, retrieval of inter-feature relationships, and higher-level analysis functionality. Queries fall broadly into two categories: *steering* and *calibration*. Steering queries identify interesting portions of the input space that correspond to observed phenomena, which helps orient analysts to particular portions of the dataset. Calibration queries refine and adjust the results of steering queries, enabling constraints to be relaxed or preprocessing such as normalization to be performed. During these retrievals, we attempt to minimize I/O costs by reducing the size of results and ensuring frequently-used records are cached by the system.

Example queries in the following sections are direct, textual representations of our Python-based TRIDENT shell (including syntax highlighting). The shell enables analysts to issue and fine-tune queries incrementally, plot and manipulate results, and transfer training data directly to other environments, such as Spark [1].

5.1 Steering Queries

Steering queries help orient the analyst and locate particular portions of the feature space. For instance, an analyst may be interested in identifying portions of the feature space where

a particular entity's states may be considered below average, average, or extremely high. In the case of our disease spread simulation, this may require identifying portions of the feature space that correspond with prolonged disease durations. The following steering queries are supported by TRIDENT:

Relational Queries aim to discover how features interact based on their relationships. These queries may be expressed as specific values or ranges, and can include wildcards to retrieve data points that match the query constraints.

```
inputs("vaccination_priority", "latent_period")
  .where(outputs["outbreak_duration"] > 40
    and outputs["num_air_infections"] < 5)
```

Event Frequency Queries steer analysis towards data points that may be either anomalous or very common. As discussed previously, analytic base trees employ a dynamic quantization scheme to optimize for memory constraints. The dynamic bins generated by TRIDENT allow rare events to be discovered, which are placed in the coarsest-grained bins. The inverse is also true for common events, which are placed in fine-grained bins. Event frequencies are accessed with the `feature.freq()` function.

Correlation Queries can be used to discover features that exhibit correlations. Such features may indicate avenues for pruning the feature space to reduce overall dimensionality, or could reveal unintuitive relationships between variables. As an example, consider retrieving summaries for input variables when the disease outbreak duration is negatively correlated with vaccine stockpiles (provided by the `pcc` function):

```
inputs().where(
    outputs["outbreak_duration"]
      .pcc(inputs["vaccine_stockpiles"]) < -0.25)
  .summarize()
```

Joint Probability Queries retrieve the probabilities of particular values or events occurring at the same time, and are returned as multidimensional probability density functions. For example, the probability of rain is likely to increase as the amount of cloud cover increases. Joint probabilities are requested using the `feature.pdf(...)` function.

5.2 Calibration Queries

Steering queries lay the groundwork for calibration queries that enable incremental adjustment across features for further analysis. Calibration queries include:

Fuzzy Queries allow analysts to specify the approximate number of training data samples that must be retrieved. TRIDENT incrementally relaxes constraints specified alongside different features (based on their correlations with the output space) to retrieve data of interest. This is achieved by traversing neighboring vertices within the analytic base tree; the following query locates inputs that occur when the output duration is more than a standard deviation away from the mean, and ensures at least 10,000 records are retrieved by relaxing the constraint if necessary:

```
duration = outputs["outbreak_duration"]
inputs().where(
   duration > duration.mean() + duration.std()
   or duration < duration.mean() - duration.std())
.fuzzy(10000)
```

Sampling Queries facilitate sampling data from portions of the feature space. The sampling can either be uniform, where all data is considered, or *stratified* where the representation of rare values is increased. When sampling uniformly, in-memory reservoir samples are used if enough information is available. Sampling is generally driven by a requested percentage of the overall data, but TRIDENT can also sample based on query time constraints, in which case the sample size is also returned with the query results.

```
sample(0.3, method='stratified', timeBound=None)
```

Pruning Queries allow query results to be sorted or ranked. Specifically, consider the case where the input feature space has 2000 dimensions and the output space has 10 variables; an analyst may be interested in analyzing the top 25 correlated features for a particular output. In such situations, pruning the feature space reduces the number of dimensions by about 90%.

Normalization Queries leverage our normalization preprocessing functionality to ensure data is normalized for modeling purposes. Models are often more consistent when features are normalized because variations in the data are accounted for; the `feature.normalize(range=[-1, 1])` method produces normalized values without inspecting the entire dataset by leveraging summary statistics stored in the ABT.

5.3 Query Performance Evaluation

During retrieval, vertices in the ABT are evaluated and removed if they do not match the query. This process incrementally reduces the search space until only matching vertices are left. Once scope reduction is complete, the subtree itself can be serialized and transferred to the client, or metadata such as the summary statistics or file pointers can be merged into an aggregate query response. To demonstrate resolution times, we submitted randomized queries across valid feature ranges that retrieved online sketch data, subtrees, and files from the system. Table 3 contains average query resolution times for each of these query classes evaluated over our epidemiological dataset; in this benchmark, query resolution is considered complete when the data is serialized and ready to be transferred across the network. For the disk and reservoir retrievals, each file was approximately 1 MB.

TABLE 3
Query evaluation microbenchmark for each query class (averaged over 100 iterations).

Query	Time (ms)	Std. Dev.
Online Sketch	0.002	0.001
Tree-Based	61.699	2.015
Disk (100 files)	1,168.199	64.695
Reservoir (100 files)	0.131	0.100

To evaluate the end-to-end performance of TRIDENT, we issued steering and calibration queries (correlation and normalization, respectively) across both of our test datasets. Both operations were applied across the entire datasets, including all features. For a point of reference, we also implemented identical transformation algorithms in Hadoop and

Spark, using built-in functionality and optimizations where possible (in the case of Hadoop, we also leveraged our online summary statistics implementation). Table 4 contains the results of this benchmark. One contrast between the approaches is generating a correlation matrix across all the features in the dataset; our in-memory ABT already contains this information for any specified combination of features, so results take around 1 ms to produce. Normalization also benefits from the ABT because the minimum and maximum values are already available, whereas both Hadoop and Spark require an initial pass over the selected data before they can begin the normalization process. The difference in computation times between the two datasets (epidemiological and atmospheric) was largely a function of their size and feature counts. Compared to Spark, TRIDENT provides a 93% reduction in execution time when normalizing a dataset. Finally, if only a sample of the entire dataset is required, query times are decreased by drawing from in-memory reservoirs.

TABLE 4
End-to-end query tests, with similar computations executed in both Hadoop and Spark for reference. TRIDENT results include the *Epi*demiological and *Atm*ospheric datasets, as well as results from an in-memory *Res*ervoir sample. Computation times reported in seconds.

Implementation	Correlation Matrix (s)	Normalize (s)
Hadoop (Epi)	461.3	1573.3
Spark (Epi)	718.6	1129.7
Trident (Epi)	0.002	79.1
Trident (Epi-Res)	0.002	4.2
Hadoop (Atm)	228.4	750.1
Spark (Atm)	318.9	481.7
Trident (Atm)	0.002	32.4
Trident (Atm-Res)	0.002	1.1

5.4 Bias-Variance Decomposition

The *bias-variance trade-off* refers to assumptions made by models that influence relationships between inputs and outputs [28]. An optimal machine learning or statistical model should accurately capture patterns or behaviors in its training data while also generalizing to new, unseen data points. In practice, models tend to optimize for one case or the other, leading to high bias or variance. When bias is high, the model does not fully capture the patterns in the data, which leads to *underfitting*. Conversely, a model with high variance is more susceptible to noise in the data, which can result in *overfitting*. Figure 7 illustrates this trade-off. Bias and variance are derived from the mean squared error (MSE) of the model. The relationship between mean squared error, bias, and variance is demonstrated by the following equation, where the model function $\hat{f}$ approximates an unknown function f:

$$MSE(\hat{f}) = Bias(\hat{f})^2 + Var(\hat{f}) + \epsilon$$

A final term, ϵ, represents the inherent noise or *irreducible error* involved when modeling the function f. Estimating the bias and variance of a particular model requires multiple predictors to be generated and evaluated based on their

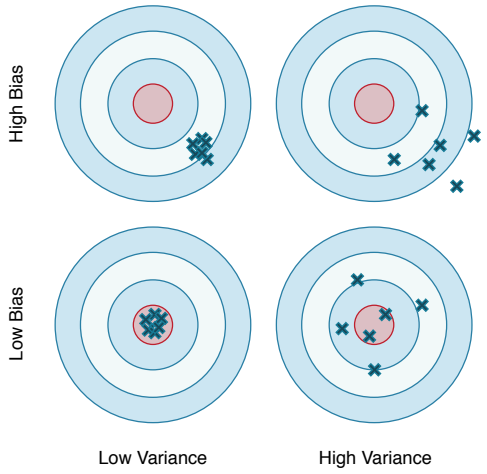


Fig. 7. Illustration of the bias-variance trade-off. An ideal predictor minimizes both bias and variance (shown in the lower left corner).

average forecasts. However, the entire training set is generally used to create a single model. To generate several new, representative training sets, we use *bootstrap aggregation*, which samples uniformly from the original dataset with replacement [28].

TRIDENT automates bias-variance decomposition to help guide analysts during model selection and parameterization. To begin the process, the analyst selects records of interest (both inputs and outputs) via queries. After the model space has been defined, reservoir samples are used during bootstrap aggregation to help avoid disk I/O. This produces a configurable number of training sets for evaluation, which can then be transferred to a computation engine or machine learning library. TRIDENT also offers three built-in models to allow the entire bias-variance decomposition process to occur within the system: *multiple linear regression* [29], *random forests* [30], and *gradient boosting* [23], [24]. Linear regression provides a baseline measure of model performance that can be trained quickly, providing initial guidance towards further modeling efforts. Random forests are often useful in restricting variance, while gradient boosting can be used to limit bias. By default, model parameters are tuned for training speed, although analysts can also customize the models that are generated. Figure 8 contains the bias-variance decomposition of each model for our particular dataset; gradient boosting provides the best predictive performance with a root-mean-square error (RMSE) of **1.78** (3.17 MSE, 3.02 bias, 0.15 variance).

5.5 Exporting Training Data

After preprocessing and selecting input and output variables, TRIDENT passes the resulting datasets to client applications. Internally, collections of data points are represented as matrices that are designed to be compatible with DataFrame implementations provided by R [8], the Python Pandas library [9], and Spark SQL [31]. A set of *adapters* allow TRIDENT datasets to be transformed to several other formats. Besides DataFrames, the current set of adapters supported by TRIDENT include (1) Spark RDDs [6], LabeledPoints, and Matrices; (2) LIBSVM [7]; (3) TensorFlow [3]

records; and (4) file-based formats such as JSON, CSV, and whitespace-delimited values. We also provide an adapter interface to allow other formats to be added to the system.

Table 5 contains the conversion speeds for each format; as one may expect, in-memory transfers to formats such as resilient distributed datasets (RDDs) tend to be faster than file-based formats such as LIBSVM [7], JSON, or CSV. However, many existing machine learning libraries are designed around file-based access. To facilitate in-memory transfers, TRIDENT supports *bulletins*, which are memory-resident datasets that can be accessed directly using a virtual FUSE [32] file system. This allows analysts to select and preprocess data, store it in a compatible file format in memory, and then train on the files with a variety of existing machine learning frameworks. If the amount of data is too large to fit into memory, feature vectors are evicted on a most recently used (MRU) basis to reflect the iterative nature of machine learning algorithms [33].

TABLE 5
Conversion speeds (in records/s) for the various export formats supported by TRIDENT.

Format	Records/s	Std. Dev.
RDD (LabeledPoint)	8,462,170	1,394
Spark Dataframe	4,519,774	1,451
CSV	1,165,883	641
LIBSVM	760,253	2,169
JSON	479,738	1,900
TensorFlow Record	97,560	140

During the export phase, TRIDENT supports creation of cross validation folds. K-fold cross validation is often used to assess the generality of a model by dividing the training data into k discrete folds, where one of the folds is used for testing and the remaining $k-1$ folds are used for training. The process repeats k times, ensuring that each fold is used once as the test set. After cross validation is complete, the average error across all k folds is computed and used to evaluate model performance. Cross validation folds are exported either as (1) separate files, or (2) a single dataset with metadata tags indicating fold membership.

5.6 Analytics Evaluation

To assess the TRIDENT analytics pipeline, we designed three experiments to forecast the outbreak duration of foot-and-mouth disease in Texas, USA under a variety of scenarios. The length of an outbreak is generally one of the most requested output features, as it has planning implications and could measure potential economic consequences.

5.6.1 Spark

Our first benchmark leveraged Spark [1] and its implementation of random forests provided by MLlib [34]. We used TRIDENT to select outbreak data stored across our 75-node cluster that exhibited disease durations longer than 15 days and occurred via airborne spread. The data points were normalized and drawn from in-memory reservoirs for a total dataset size of 25,000 records, which was then split into 10 folds for cross validation. Using MLlib, we created a random forest model parameterized to generate 100 decision trees with a maximum depth of 10. Figure 9 demonstrates the performance of the predictor, with a root-mean-square error of 1.83 days. The TRIDENT shell commands that were used to produce the model are shown in the following code listing:

```
training_folds = inputs()
  .where(inputs["spread_type"] == "airborne"
    and outputs["disease_duration"] > 15)
  .prune(outputs, 600)
  .join(outputs)
  .normalize()
  .toRDD(cvFolds=10, parallelize=True)

for fold in training_folds:
  model = RandomForest.trainRegressor(fold, ...)
```

Table 6 provides a breakdown of the steps taken to generate the model and their respective latencies, averaged over 100 iterations. Note that training jobs were dispatched by Spark across 24 nodes in the cluster, and the model training time reported was averaged across all 10 cross validation folds within the test iteration.

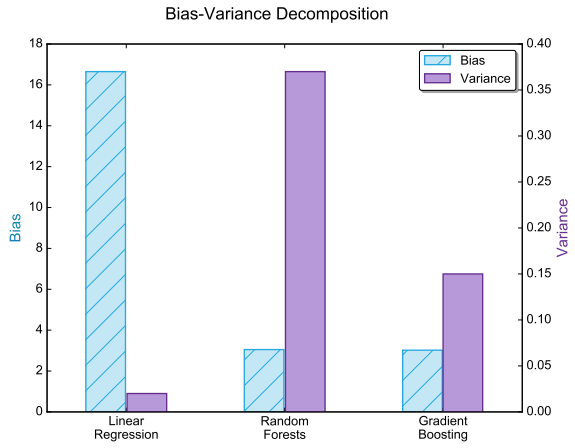


Fig. 8. Bias-variance decomposition for three of the built-in models supported by TRIDENT.

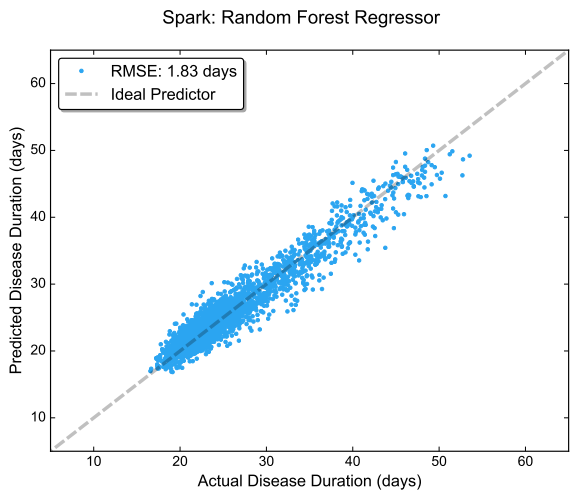


Fig. 9. Prediction performance of a random forest model generated in Spark with our RDD export functionality.

TABLE 6
Timing information for each step in producing random forest models with Spark, averaged over 100 runs.

Step	Completion (s)	Std. Dev
Query	1.21	0.11
Normalization	0.01	0.01
Format Conversion	0.18	0.01
Model Training	41.16	1.50

5.6.2 TensorFlow

To further test our export functionality, we converted the same dataset used in our Spark experiment to TensorFlow [3] records and applied the TensorFlow linear regression implementation with a gradient descent optimizer. Figure 10 demonstrates the performance of the predictor over 10 cross-validation folds, with a root-mean-square error of 3.80 days. In this case, the linear model limits prediction accuracy (especially for longer-lasting disease outbreaks). However, linear models often train faster and can be used during the exploration process to evaluate feature selection.

5.6.3 Distributed Ensembles: scikit-learn

Another modeling possibility supported by TRIDENT is the generation of local model instances in parallel using the datasets available at each storage node, creating a *distributed ensemble* that trains quickly and specializes for each unique portion of the feature space. We developed a machine learning application based on the scikit-learn library [4] and configured TRIDENT to launch it once training data was available and stored in memory-resident bulletins. To assess the trade-off space associated with the distributed ensemble, we also built monolithic models by sampling across the entire dataset. Table 7 reports the root-mean-square error (RMSE) and coefficient of determination (R^2) for each of these tests on multiple linear regression (LR), random forest (RF), and gradient boosting (GB) models.

As demonstrated by this evaluation, the gradient boosted decision tree models outperform multivariate linear regression. This is likely due to the linear model not fitting the data. Another insight gained from this experiment was that building many localized models in parallel can be highly effective; using the same number of feature vectors, most of our local models achieved higher predictive performance than a global sample of the same size. On the other hand, the model with 75,000 samples demonstrates the effectiveness of a general model built with a much larger training corpus. Each of these models represents a different portion of the analytics trade-off space; linear regression is simple and fast to train but can have limited accuracy, while building specialized models can be advantageous for ad hoc modeling.

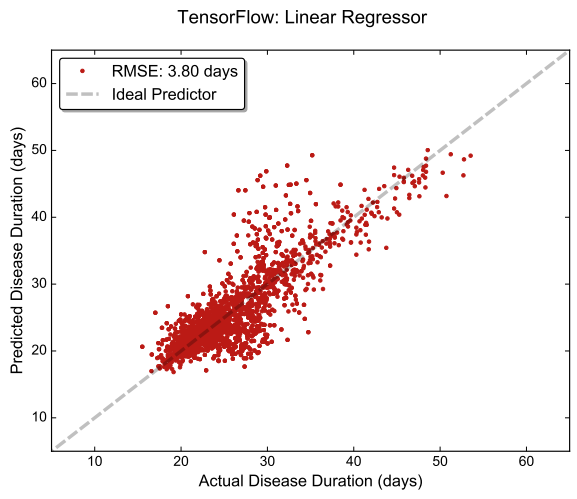


Fig. 10. Prediction performance of a linear regression model generated in TensorFlow with our TFRecord export functionality.

TABLE 7
Statistics for analytic models produced by TRIDENT, which were computed with 10-fold cross-validation.

Model	Scope	Data Points	RMSE	R^2
LR	Local	12,000	4.08	0.63
RF	Local	12,000	2.34	0.87
GB	Local	12,000	1.78	0.93
LR	Global	12,000	3.86	0.66
RF	Global	12,000	2.42	0.85
GB	Global	12,000	1.90	0.92
GB	Global	75,000	1.57	0.95

6 RELATED WORK

Spark [1] is a cluster computing framework that supports memory resident datasets and computations described by directed, acyclic graphs (DAGs). This allows expensive disk accesses to be avoided while also facilitating graph-based workflows and iterative applications such as machine learning algorithms. One of the key features of Spark is its Resilient Distributed Datasets (RDDs), which provide in-memory parallel data manipulation functionality [6]. Spark can process data from a variety of underlying storage services, but lacks the integrated storage and exploration facilities implemented in TRIDENT. MLBase [35] extends the Spark stack to support machine learning tasks with RDDs. MLlib [34] contains a number of machine learning algorithms, and an integrated ML Optimizer assists users in finding the best parameters for their models.

Hadoop [2] and HDFS [5] provide a scalable solution for managing and processing large datasets. While HDFS does not partition records based on their content, the software is widely available and compatible with several distributed computation engines. Hadoop provides distributed batch processing capabilities, but native support for real time exploration is limited unless extensions such as HaLoop [36] are used. Additionally, while the computational facilities in Hadoop could be leveraged to perform data preprocessing, intermediate records would need to be written to the disk and stored as full-resolution blocks.

Cassandra [15] is a DHT-based storage and data processing system that also supports custom partitioning algorithms. While implementing a cluster-based partitioning scheme would be possible in Cassandra, the system employs a wide-column storage format similar to BigTable [37],

which limits indexing efficiency over datasets with high dimensionality. Cassandra supports CQL (Cassandra Query Language) for retrieval operations and MapReduce computations for data manipulation and analysis, but does not offer approximate, ad hoc query capabilities or preprocessing focused on machine learning algorithms.

Synopsis [38] builds compact sketches over multidimensional data streams to provide approximate representations of large datasets. While this approach could be adapted for machine learning and analysis in a similar fashion to TRIDENT, the system is designed for in-memory storage rather than incorporating large-scale persistent storage.

Galileo [39], [40], [41] employs a network design and indexing scheme similar to TRIDENT. However, Galileo generally handles tens of features rather than the thousands of features produced by the inputs and outputs of discrete event simulations. This changes the trade-off space associated with efficiently indexing and managing the data, especially with in-memory records; while Galileo uses an iterative approach for reconfiguring its index incrementally, our KDE-based quantization requires less frequent reconfiguration, and therefore does not require as much state synchronization between nodes. Galileo also inspects and partitions its datasets based on geospatial properties, which is not advantageous for model building in our use case.

BlinkDB [42] provides approximate query processing functionality based on Hive [43]. This allows analysts to query large datasets while specifying time and error bounds. BlinkDB supports high-level summaries created by sampling randomly across the entire dataset, along with finer-grained stratified samples over frequently-accessed records. This approach facilitates exploratory search over precomputed dimensions of interest, but does not offer the ad hoc queries or preprocessing provided by TRIDENT.

7 Conclusions and Future Work

This study describes TRIDENT, which supports analytics over voluminous data by: (1) distributing data and query evaluations, (2) supporting a rich set of preprocessing operations that simplify identification of important features, (3) incorporating support for a set of compact, memory-resident data structures that are amenable to fast querying, and (4) supporting an expressive set of queries to explore the feature space and retrieve/convert data.

TRIDENT controls the placement of incoming feature vectors by reducing their dimensionality and clustering similar data points. Cluster quality is evaluated with the Davies-Bouldin index, and we demonstrate improvements in building specialized local models across the nodes in the system (**RQ1**). After partitioning, feature vectors are passed to online sketch instances and our memory-resident, hierarchical analytic base tree (ABT) data structures. This allows information to be retrieved about the underlying dataset and transformations (such as normalization or correlation analysis) to be applied without requiring disk I/O (**RQ2**). Additionally, our analytic base trees support flexible queries to locate and refine portions of the feature space in memory. Online summary statistics also provide detailed information about the features under study without accessing files on disk, and preprocessing operations are cached to reduce duplicate transformations (**RQ3**). Finally, our query-driven approach allows subsets of the feature space to be selected, creating training data sets that can be passed on to machine learning frameworks. To support such activities, we provide a base set of analytical models that can serve as pilot studies. Bias-variance decomposition of these models is also made available to allow the analyst to judge performance (**RQ4**).

As part of our future work we plan to incorporate support for visualizations. Specifically, our goal is to support rapid explorations and both creation and assessments of multiple model instances. We envisage creation of multiple model instances (different algorithms, hyperparameters, and regular constraints) with the same training data. The key goal of the visualization process is to assist in understanding why a model instance outperforms others.

The latest version of TRIDENT can be downloaded at: http://galileo.cs.colostate.edu/trident

Acknowledgments

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

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[2] C. Lam, *Hadoop in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[3] M. Abadi, P. Barham, J. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026899

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[6] M. Zaharia *et al.*, "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*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[7] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011.

[8] R Core Team *et al.*, "R: A language and environment for statistical computing," *https://www.r-project.org/*.

[9] Pandas Developers, "Pandas python data analysis library," *http://pandas.pydata.org*.

[10] N. Harvey, A. Reeves, M. Schoenbaum *et al.*, "The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions," *Preventive Veterinary Medicine*, vol. 82, no. 3, pp. 176–197, 2007.

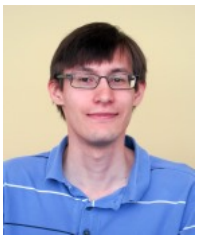
[11] National Oceanic and Atmospheric Administration. (2016) The north american mesoscale forecast system. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[12] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[13] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*. Springer, 2001, pp. 329–350.
[14] G. Manku, M. Bawa, P. Raghavan *et al.*, "Symphony: Distributed hashing in a small world," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, vol. 4, 2003, pp. 10–10.
[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[16] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.
[17] G. Niemeyer. (2008) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash
[18] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "StreamKM++: A clustering algorithm for data streams," *J. Exp. Algorithmics*, vol. 17, pp. 2.4:2.1–2.4:2.30, May 2012.
[19] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: massive online analysis," *Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, 2010.
[20] A. Zimek, E. Schubert, and H.-P. Kriegel, "A survey on unsupervised outlier detection in high-dimensional numerical data," *Statistical Analysis and Data Mining*, vol. 5, no. 5, pp. 363–387, 2012.
[21] D. L. Davies and D. W. Bouldin, "A cluster separation measure," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 224–227, April 1979.
[22] L. Breiman, "Arcing the edge," Technical Report 486, Statistics Department, University of California at Berkeley, Tech. Rep., 1997.
[23] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.
[24] L. Mason, J. Baxter, P. Bartlett, and M. Frean, "Boosting algorithms as gradient descent in function space." NIPS, 1999.
[25] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
[26] M. Kristan, A. Leonardis, and D. Skocaj, "Multivariate online kernel density estimation with gaussian kernels," *Pattern Recognition*, vol. 44, no. 10-11, pp. 2630–2642, 2011.
[27] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.
[28] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics Springer, Berlin, 2001, vol. 1.
[29] D. A. Freedman, *Statistical models: theory and practice*. cambridge university press, 2009.
[30] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324
[31] M. Armbrust, R. S. Xin, Lian *et al.*, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.
[32] M. Szeredi *et al.*, "File system in user space (FUSE)," *https://github.com/libfuse/libfuse*, 2016.
[33] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, ser. VLDB '85. VLDB Endowment, 1985, pp. 127–141. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286760.1286772
[34] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "MLlib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, Apr. 2016.
[35] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "Mlbase: A distributed machine-learning system." in *CIDR*, vol. 1, 2013, pp. 2–1.
[36] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920881
[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.
[38] T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara, "Synopsis: A distributed sketch over voluminous spatiotemporal observational streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2552–2566, Nov 2017.
[39] M. Malensek, S. L. Pallickara, and S. Pallickara, "Analytic queries over geospatial time-series data using distributed hash tables," *IEEE Transactions on Knowledge and Data Engineering*, vol. PP, no. 99, pp. 1–1, 2016.
[40] ——, "Autonomously improving query evaluations over multidimensional data in distributed hash tables," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC)*, Sep 2013, pp. 15:1–15:10.
[41] ——, "Evaluating geospatial geometry and proximity queries using distributed hash tables," *IEEE Computing in Science Engineering (CiSE)*, vol. 16, no. 4, pp. 53–61, Jul 2014.
[42] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42.
[43] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687553.1687609

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

Walid Budgaga is a Ph.D. candidate in the Department of Computer Science at Colorado State University. His research involves distributed analytics, anomaly detection, and scalable computation.
Email: wbudgaga@cs.colostate.edu



Ryan Stern is a Ph.D. candidate in the Department of Computer Science at Colorado State University. His research interests are in the area of visual analytics with an emphasis on generating real-time views of voluminous datasets. This involves coping with issues such as representativeness, memory residency, and page-fault avoidance.
Email: rstern@cs.colostate.edu

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

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