

Splash: Ad-Hoc Querying of Data and Statistical Models

Lujun Fang, Kristen LeFevre

Electrical Engineering and Computer Science, University of Michigan
2260 Hayward Ave, Ann Arbor, MI 48109

ABSTRACT

Data mining is increasingly performed by people who are not computer scientists or professional programmers. It is often done as an iterative process involving multiple ad-hoc tasks, as well as data pre- and post-processing, all of which must be executed over large databases. In order to make data mining more accessible, it is critical to provide a simple, easy-to-use language that allows the user to specify ad-hoc data processing, model construction, and model manipulation. Simultaneously, it is necessary for the underlying system to scale up to large datasets. Unfortunately, while each of these requirements can be satisfied, individually, by existing systems, no system fully satisfies all criteria.

In this paper, we present a system called *Splash* to fill this void. *Splash* supports an extended relational data model and SQL query language, which allows for the natural integration of statistical modeling and ad-hoc data processing. It also supports a novel *representatives* operator to help explain models using a limited number of examples. We have developed a prototype implementation of *Splash*. Our experimental study indicates that it scales well to large input datasets. Further, to demonstrate the simplicity of the language, we conducted a case study using *Splash* to perform a series of exploratory analyses using network log data. Our study indicates that the query-based interface is simpler than a common data mining software package, and it often requires less programming effort to use.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications

General Terms

Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Updated July 15, 2010

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...\$10.00

Data mining is increasingly performed by non-experts as an iterative, exploratory process involving multiple ad-hoc tasks, as well as data pre- and post-processing, all of which must be executed over large databases. One specific example of this, which we find particularly interesting, is the ad-hoc exploratory analysis of logs for misuse detection.

While recent work has begun to consider exploratory mining paradigms [19], most mining software still supports a rigidly-defined API, or set of tasks (e.g., classification with a fixed training set and class label). In contrast, relational databases have been extremely successful in providing simple languages that allow lay users to specify ad-hoc queries and data manipulations. Further, many years of research have resulted in database systems that scale to large data. The goal of this work is to develop a system that combines the power of data mining (specifically, statistical modeling) with the simple ad-hoc queries and scale of SQL and relational databases.

1.1 Motivating Example: Log Analysis

Maintaining audit logs is a fundamental component of a comprehensive security [5] and privacy [15] infrastructure. Logging is complementary to access control and other security mechanisms, and it is particularly useful for recording and detecting inappropriate access and misuse by insiders.

To illustrate the importance of audit logs, consider a health-care organization, which must take precautions to safeguard sensitive information, including patients' medical records. The organization has deployed a comprehensive security infrastructure, but due to the evolving nature of care (e.g., residents and nurses who change departments frequently), it is often impossible to specify comprehensive access control policies. In fact, overly-restrictive policies interfere with patient care. Instead, rather than *preventing* inappropriate access, it is often necessary to take steps to *detect* such access after the fact by keeping a record of what information has been accessed, and by whom [14]. As a recent example, Kaiser Permanente recently fired fifteen employees for inappropriately viewing the medical records of Nadya Suleman, the highly-publicized mother of octuplets [18].

Legislation has begun requiring organizations in domains such as healthcare to track their use of sensitive data [1], but few tools have been developed to allow auditors to systematically and proactively analyze the resulting logs.

Anomaly detection is one common approach to log analysis and intrusion detection, which has been researched extensively for operating systems [26], networks [43], and more recently for database systems [33]. At a high level, the idea is to learn a model of "normal" behavior, and then detect

deviations from normal as potential misuse. Unfortunately, even in domains such as operating systems and networks, where anomaly detection has been studied extensively, it is known to have significant disadvantages, including the bothersome problem of false positives [43], and a notable lack of flexibility. For example, an administrator must decide (a priori) the granularity at which to model behavior (e.g., user, role, etc.), and the output of the system is limited to simple boolean decisions (raise a warning or not).

While this leads us to consider a more flexible and configurable system, it also suggests a practical problem: Our target users (e.g., healthcare compliance officers) are often not trained programmers, so a system that requires significant programming effort is impractical.

1.2 System Requirements

Inspired by the log analysis problem, we set out to design a system that marries statistical modeling (following the past successes of anomaly detection) with the ad-hoc flexibility and simplicity of high-level query languages, and the scalability of relational databases. Specifically, we identified three high-level requirements:

- **High-Level Declarative Query Language:** The system should support a simple query language that allows for the specification of ad-hoc queries, including data pre- and post-processing, with minimal programming effort.
- **Ad-Hoc Model Creation / Manipulation:** The data model and query language should provide a simple and natural abstraction for ad-hoc creation and direct manipulation of one or more statistical models.¹
- **Scalability:** The system should be able to scale gracefully to large input databases, particular those that are significantly larger than main memory.

Figure 1 provides a rough evaluation of related systems according to these criteria. (A shaded circle indicates that the system satisfies the requirement.)

While statistical software (e.g., Matlab [7], R [9], Stata [12], SAS [11]) and data mining packages (e.g., Weka [13], RapidMiner [39]) provide a great deal of functionality, to the best of our knowledge they assume that the data to be analyzed is small enough to fit in main memory. SAS includes libraries for scalable data pre-processing [11], and recent work has proposed techniques that allow R to scale to larger datasets [53]. However, neither provides a high-level declarative query language, which makes ad-hoc exploratory analysis more difficult than in a typical SQL database.

For these reasons, several systems have proposed incorporating support for statistical and data mining models into relational databases and SQL. However, the proposed abstractions do not naturally lend themselves to ad-hoc model creation and manipulation.

MauveDB [22] provides an abstraction called a model-based view. The idea is to expose to the user data interpolated from the model, as if this data were part of an ordinary database view. While this is natural for data interpolation (e.g., in the sensor domain), it suffers two main shortcomings when it comes to ad-hoc analysis: (1) Each model must

¹Wikipedia describes a statistical model as a “mathematical description of the behavior of an object in terms of random variables and associated probability distributions.” Statistical models can be used in a variety of mining tasks, including classification, outlier detection, and others.

	Declarative Query Language	Ad-Hoc Model Creation / Manipulation	Scalability
Statistical Software (e.g., Matlab, R, Stata, etc.)	○	◐	○
Data Mining Packages (e.g., Weka)	○	◐	○
MauveDB	●	○	●
Microsoft DMX, Ole DB for Data Mining	●	○	●
IBM Intelligent Miner	◐	○	●
<i>Splash*</i>	●	●	●

Figure 1: Comparison of Existing Systems

be declared using syntax similar to SQL’s CREATE VIEW, which leads to a tedious process if an analyst wants to create many models (e.g., one per system user in the log analysis domain), and (2) Aside from the interpolated views, models are completely hidden from the user. This makes it difficult, for example, to use a model-based view for the purpose of detecting anomalies (records that deviate significantly from a given model), or for comparing models.

Microsoft SQL Server’s DMX [2] and OleDb for DM [40] expose classification and regression models through an abstraction known as a prediction-join. This abstraction provides a natural way of assigning class labels to data, but it does not allow a user to compare a record to a model, or to compare two models (e.g., for the purpose of detecting anomalies). Also, like MauveDB, the user must specify each model separately using extended SQL DDL, which can be cumbersome. IBM’s Intelligent Miner [3] provides scalable data mining operations, which can be applied to data residing in a relational database, but models must be specified one-by-one, either using PMML (a markup language) or the graphical interface.

1.3 Paper Overview

In response to these requirements, we have designed and built a system called *Splash*. *Splash* supports a set of simple extensions to the relational data model and SQL query language (Section 2). These extensions allow for the ad-hoc creation and manipulation of models, and they also provide a “workbench” that allows the user to leverage the full power of SQL when creating and applying the models. To further assist users in understanding and interacting with models, we have also defined and implemented a novel *representatives* operator, which is used to explain a model using a limited number of examples (Section 3). Section 4 describes the details of our prototype, including performance optimizations.

An extensive experimental study (Section 5) evaluates *Splash* with respect to our requirements. In particular, we evaluated performance and scale. Also, we conducted a case study using network attacks logs in order to evaluate the effectiveness of the proposed data model and query language. We found that ad-hoc analyses are often more easily expressed using the query language of *Splash* than using a common open-source data mining package.

2. DATA MODEL & QUERY LANGUAGE

The data model and query language for *Splash* are based on simple extensions to the relational model and SQL, which

incorporate support for statistical models as a new data type (like an integer or string). Ad-hoc model construction is naturally viewed in terms of a new aggregate function (similar to SUM or AVG), and models are naturally manipulated using a set of primitive similarity functions.

2.1 Basics

The basic extensions to the relational data model are generally quite simple, yet powerful. They are based on two basic data types: the *feature vector* and the *profile*, as well as a new *profile aggregation operator*.

- **Feature Vectors:** A *feature space* is defined by n variables X_1, \dots, X_n , which can alternately be viewed as defining an n -dimensional space. A *feature vector* is a point $\langle x_1, \dots, x_n \rangle$ in this space. (For clarity, we will use capital letters to denote feature / variable names, and lower-case letters to denote instances, or feature values.)
- **Profiles:** A *profile* is an estimated joint probability density function (pdf) over a particular feature space.² We will denote a profile constructed over feature space $\langle X_1, \dots, X_n \rangle$ as $\hat{f}_{X_1, \dots, X_n}(x_1, \dots, x_n)$.
- **Profile Aggregation Operator:** A *profile aggregation operator*, denoted $profile(D)$ is an aggregate function that takes a set of feature vectors D (drawn from the space $\langle X_1, \dots, X_n \rangle$) as input, and produces a profile. The operator is also easily extended to first partition D based on some other database attribute P , and construct one profile per unique value p of P . (This is similar to a SQL GROUP BY query.)

These basic building blocks are nicely integrated with standard SQL, as illustrated by the following example:

EXAMPLE 2.1. Consider the simple example shown in Figure 2. On the left is the relation `AUDIT_LOG_FEATURES`, which contains four attributes. The final attribute contains salient features describing the transaction. (For illustrative purposes, we show two features: query type, and number of records affected.) In practice, the feature vector can be constructed for each tuple by defining an application-specific feature-extraction function (e.g., as a UDF). Much data processing is easily specified in SQL and existing extensions.

Taking `AUDIT_LOG_FEATURES` as input, it is easy to construct profiles on an ad-hoc basis. For example, the following query constructs one profile per `User_ID` value (as shown in the figure):

```
SELECT User_ID, profile(Features)
FROM AUDIT_LOG_FEATURES
GROUP BY User_ID
```

2.2 Query Language

While the profile aggregation operator provides easy ad-hoc model creation, we also want to provide primitives for direct interaction with, and manipulation of, models. We find that many manipulations can be expressed simply by exposing two primitive similarity functions:

- **Similarity between feature vector and profile:** The similarity between a feature vector and a profile, denoted $sim(\langle x_1, \dots, x_n \rangle, p)$, is a real number in $[0, 1]$.

²In practice, we expect a combination of discrete and continuous features; the generalization is straightforward.

- **Similarity between two profiles:** The similarity between two profiles is denoted $sim(p_1, p_2)$, and is also a real number in $[0, 1]$.

EXAMPLE 2.2. Continuing with the running example, the combination of profile aggregation and similarity allows us to perform a variety of useful tasks. For example, we can easily specify the query requesting anomalous log records, defined as any log record such that the similarity between the record and its user's profile is less than a given threshold:

```
SELECT F.RID
FROM AUDIT_LOG_FEATURES F,
     (SELECT profile(Features), User_ID
      FROM AUDIT_LOG_FEATURES
      GROUP BY User_ID) AS Profiles(P, User_ID)
WHERE F.User_ID = Profiles.User_ID
AND sim(F.Features, P) < threshold
```

Instead of profiling behavior at the level of individual users, this query can be modified to construct profiles in another way, for example using another attribute `Role_ID`. It is also easy to alter the query to request a ranked list of potentially anomalous records by removing the threshold and adding `ORDER BY sim(Features,P) ASC` to the query.

EXAMPLE 2.3. Continuing with the log analysis example, we can also construct a query to discover users whose behavior has changed significantly since the previous month:

```
SELECT Profile1.User_ID
FROM (SELECT profile(Features), USER_ID
      FROM AUDIT_LOG_FEATURES
      WHERE MONTH = 'May'
      GROUP BY User_ID) AS Profile1(P, User_ID),
     (SELECT profile(Features), User_ID
      FROM AUDIT_LOG_FEATURES
      WHERE Month = 'June'
      GROUP BY User_ID) AS Profile2(P, User_ID)
WHERE Profile1.User_ID = Profile2.User_ID
AND sim(Profile1.P, Profile2.P) < threshold
```

2.3 Sample Instantiation

The basic primitives outlined in Sections 2.1 and 2.2 are very general, and they can be instantiated in countless ways. Our main objective is to provide a clean abstraction for integrating SQL and probabilistic models, not to develop new machine learning techniques. Nonetheless, we briefly describe a sample profile aggregation operator and similarity functions, which we have implemented in our prototype.

2.3.1 Sample Profile Aggregation Operator

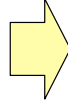
There are countless ways to estimate $\hat{f}_{X_1, \dots, X_n}(x_1, \dots, x_n)$ from D . One profile aggregation operator uses the simplifying assumption that X_1, \dots, X_n are independent (optionally conditioned on group-by attribute P).³ That is, assume that $f_{X_1, \dots, X_n}(x_1, \dots, x_n) = \prod_{i=1}^n f_{X_i}(x_i)$, and estimate each $\hat{f}_{X_i}(x)$ separately.

When X_i is discrete-valued, f_{X_i} is a probability mass function, and it is easy to estimate using counts: $\hat{f}_{X_i}(x) = \frac{|\{d \in D: d.X_i=x\}|}{|D|}$. (To avoid the case where some counts are zero, a simple adjustment adds one to the numerator and denominator.) When X_i is continuous, we estimate the probability density function using a Gaussian kernel density estimator. If h is a smoothing parameter and $K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$, then we define $\hat{f}_{X_i}(x) = \frac{1}{|D| \cdot h} \sum_{d \in D} K(\frac{x-d.X_i}{h})$.

³This is closely related to the Naive Bayes assumption.

AUDIT_LOG_FEATURES

RID	User_ID	Month	Features
1	alice	January	<select, 250>
2	alice	January	<select, 2>
3	alice	March	<update, 1>
4	bob	June	<insert, 25>
5	bob	August	<select, 103>



User_ID	Profile
alice	profile(<select, 250>, <select, 2>, <update, 1>)
bob	profile(<insert, 25>, <select, 103>)

Figure 2: Profile Aggregation Example

2.3.2 Sample $sim()$ Functions

- $sim(\langle x_1, \dots, x_n \rangle, p)$: Suppose that profile p is defined by estimated distribution $\hat{f}_{X_1, \dots, X_n}$. One approach is to let $sim(\langle x_1, \dots, x_n \rangle, p) = \hat{f}_{X_1, \dots, X_n}(x_1, \dots, x_n)$. When one or more of the features X_i is continuous, a simple adjustment integrates f_{X_i} over an interval $(\pm\delta)$ surrounding x_i .
- $sim(p_1, p_2)$: Suppose that profiles p_1 and p_2 are defined, respectively, by $\hat{f}_{X_1, \dots, X_n}$ and $\hat{g}_{X_1, \dots, X_n}$. The KL-divergence is a common information-theoretic way of measuring the difference between two distributions:⁴

$$D_{KL}(\hat{f}_{X_1, \dots, X_n} \parallel \hat{g}_{X_1, \dots, X_n}) = \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} \hat{f}(x_1, \dots, x_n) \log \frac{\hat{f}(x_1, \dots, x_n)}{\hat{g}(x_1, \dots, x_n)} dx_1 \dots dx_n$$

We can use this to construct a similarity function:

$$sim(p_1, p_2) = \frac{1}{1 + D_{KL}(\hat{f}_{X_1, \dots, X_n} \parallel \hat{g}_{X_1, \dots, X_n})}$$

3. PROFILE EXPLANATION & FINDING REPRESENTATIVES

Aggregation provides a natural way of creating profiles, and the $sim()$ functions allow basic interaction with profile objects. However, the user may want to further understand the meaning of certain profiles. For example, in log analysis, a system administrator may have detected an abnormal profile, which he would like to understand better.

As one means of explaining a profile, we propose an operator that, given a profile p and dataset D , selects a small subset of D to *represent* the profile distribution. (To keep the operator as general as possible, we do not require that D be a sample from the same distribution as p .) This results in the following optimization algorithm. While related problems have been considered in the literature (see Section 6), to the best of our knowledge, this problem has not been addressed.

DEFINITION 1 (REPRESENTATIVE SET). *Given a profile p , a set of feature vectors D , and user-defined parameter k , the optimal representative set, denoted $rep(p, D, k)$, is the set $D' \subseteq D$ such that $|D'| \leq k$ and $sim(p, profile(D'))$ is maximized.*

In the above, $profile(D')$ represents the pdf derived from the points in D' after applying kernel density estimation, and $sim(p_1, p_2)$ is the same similarity function described earlier.

3.1 Algorithms

Due to the combinatorial nature of the problem, we have developed several heuristics for selecting representative sets.

⁴Of course, KL-divergence is not symmetric, so a common trick computes both values and takes the average.

In the following descriptions, D consists of m feature vectors, each of which is a point in n -dimensional space.

Sampling Algorithm (RAND-R): The simplest (naive) approach is to choose a simple random sample from D . While sampling is widely used in statistics, there are two clear problems to using this approach here. First, while $profile(D')$ will converge to p (for large k) if D is drawn from the distribution p , RAND-R is only effective in this case. When $profile(D) \neq p$, $profile(D')$ converges to $profile(D)$ rather than p . Second, even when RAND-R is guaranteed to converge for large k , we would ideally like to choose a subset that is as small as possible.

Histogram-Based Sampling (HIST-R): To overcome the shortcomings of RAND-R, one approach is to actively allocate the positions of representative points. This can be done by partitioning the space into subregions, and allocating representatives to each subregion based on the region's probability mass in p .⁵

To partition the space, we leverage the well-known multi-dimensional histogram partition rule MHIST [44] and the partition constraint V-optimal [32]. The partition algorithm is recursive; it begins by dividing the whole space into two regions, and then recursively partitions each of the resulting regions until a stopping criterion is met. Each iteration proceeds as follows. (Suppose that we are working on subregion R , which contains m^* points from D , and based on probability mass, we need to select k^* representative points from this subregion).

1. Select a dimension to divide. (The dimension with maximum variance in p is selected, following the MHIST heuristic.)
2. Choose the point at which to divide the selected dimension. (We choose the point according to the V-optimal criterion.)
3. Divide the m^* points between the resulting subregions R_1 and R_2 , and set k_1^* and k_2^* (the number of representatives that need to be chosen from R_1 and R_2) according to the probability mass of p in R_1 and R_2 .

The partitioning process continues recursively until $m^* = k^*$ or $k^* = 1$, at which point we randomly select k^* points from those in D that fall in the subregion (R).

There are $O(k)$ iterations. In each iteration, Step 1 takes $O(n)$ time, Step 2 takes $O(c)$ time where c is the number of distinct values in the chosen dimension, Step 3 takes $O(m)$ time. Therefore the running time for the algorithm is $O((c + n + m)k)$.

While this approach eliminates some of the problems of RAND-R in that it can be used when D and p do not have

⁵Suppose that a space partitioning algorithm divides the space into l subregions $\{R_1, R_2, \dots, R_l\}$. For the region R_i ($1 \leq i \leq l$), we need to pick $k^* \int_{R_i} p$ points as representatives for that region.

the same distribution, it is also dependent on the partitioning algorithm. In high-dimensional space, histograms are known to deteriorate in quality. (This is confirmed by our experiments.) Thus, we propose a third and final algorithm.

Greedy Algorithm (GREEDY-R): To overcome the limitations of HIST-R, we propose a third algorithm called GREEDY-R, which greedily selects the next most representative point.

Remember that a kernel density estimation based on k points just sums up the k kernels, using a weight $w = \frac{1}{k}$ for each. The process of the greedy algorithm is to incrementally add representative points into D' ; for each new point d , we incrementally maintain a *partial* profile p' by adding the kernel for d using weight $w = \frac{1}{k}$. (For $|D'| < k$, p' is not a true probability distribution.) The difference between p and p' at d is denoted $\delta(p, d) = p - p'$.

The greedy algorithm works as follows: At each step, we choose the point d from $D - D'$ that maximizes $\delta(p, d)$. This process repeats until all k representative points are chosen. The intuition behind the heuristic is that at each step we always look for points that minimize the gap between the profile p and the sum of the kernels p' . We demonstrate the effectiveness of the algorithm in Section 5.

The greedy heuristic takes k steps. At each step it takes $O(mn)$ time to pick the best point as next representative point and to update the partial distribution after a point is picked. Therefore the running time for the greedy heuristic is $O(kmn)$.

4. IMPLEMENTATION

We have constructed a prototype of Splash using extensions to PostgreSQL [8], an open-source DBMS that supports user-defined types, operators, functions, and aggregates. It is noteworthy that all of the functionality can be implemented without costly modifications to the database engine. In addition, we have implemented several performance optimizations based on materializing and compressing profiles.

We defined new types to represent profiles and feature vectors, and functions to support operations on profiles and feature vectors. All of the extensions are implemented using C. The following is a list of the new types, functions, aggregates, and operators:

- **Feature** is a new type representing a feature vector. (The dimensionality of a **Feature** instance is the length of the vector.) In the current implementation, we support feature entries of types *integer*, *varchar*, *varbit*, and *real*.
- **Profile** is another new type. In our prototype, we implement a profile using a collection of n hash tables, where n is the total number of different features. The i^{th} hash table contains all distinct values of feature X_i (the keys) and an integer count for each. Thus, the space required to store a **Profile** instance is $O(cn)$, where c is the average number of different values per feature. We use this representation for both discrete and continuous features; the differences in probability estimation (i.e., count-based or kernel density estimation) are encapsulated by the two similarity functions.
- **profile(Feature)** is a new aggregate function, which produces a **Profile** instance, given a set of **Feature** instances. In PostgreSQL, user-defined aggregates are expressed in terms of state values and state transition functions. For this reason, we have defined the transition function **profile(Profile, Feature)** to enable adding one **Feature**

instance to an existing **Profile**. Assuming that there are m **Feature** vectors, each with dimensionality n , in this implementation, the time complexity of adding one **Feature** value to a **Profile** is $O(n)$, and the time complexity of building a **Profile** from m **Feature** vectors is $O(mn)$.

Of course, this approach assumes an implementation of **Profile** that is *incrementally updatable*. This is clearly a desirable characteristic of the profile model. For other profile models, we might need to maintain additional state. In addition, the query optimizer treats queries involving profile aggregation just as it would treat any other aggregate query. For the most part, this works reasonably well, but there are some unexpected problems, as we discuss in more detail in Section 5.

- **sim(Feature, Profile)** is a function that evaluates the similarity between a **Feature** instance $\langle x_1, \dots, x_n \rangle$ and a **Profile** instance p , as described in Section 2. For discrete feature X_i , we can compute $f_{X_i|P=p}(x_i)$ in $O(1)$ time from the internal profile representation. For continuous feature X_i , we can compute $\int_{x_i-\delta}^{x_i+\delta} \hat{f}_{X_i|P=p}(x)$, in time $O(c)$, where c is the number of distinct values for X_i . Therefore, when all features are discrete, we can compute this function in $O(n)$ time; otherwise it takes $O(cn)$.
- **sim(Profile, Profile)** is a function that evaluates the similarity between two **Profile** instances using the KL-divergence as described in Section 2. We observe that, due to the conditional independence assumption, it is possible to compute the KL-divergence for each feature independently. That is, if we have two profiles defined by distributions f and g , and we assume $\hat{f}_{X_1, \dots, X_n} = \hat{f}_{X_1} \dots \hat{f}_{X_n}$ and $\hat{g}_{X_1, \dots, X_n} = \hat{g}_{X_1} \dots \hat{g}_{X_n}$, then $D_{KL}(\hat{f}_{X_1, \dots, X_n} || \hat{g}_{X_1, \dots, X_n}) = D_{KL}(\hat{f}_{X_1} || \hat{g}_{X_1}) + \dots + D_{KL}(\hat{f}_{X_n} || \hat{g}_{X_n})$.⁶

Thus, this function is straightforward for discrete features, and can be computed in $O(cn)$ time, where n is the dimensionality of the feature vector, and c is the number of distinct values per feature. When one or more of the features X_i is continuous, we handle that by discretizing the range of X_i into r discrete ranges, and then estimate the probability density within each range using the stored counts and kernel density estimator. In this case, the time complexity is $O(rcn)$.

4.1 Materialization

Aggregate materialization is commonly used to improve performance in OLAP-style data analysis. In much the same way, it may be useful to materialize profiles to improve the performance of certain workloads in Splash. For example, each of the sample queries in Section 2.2 required computing one profile per **USER_ID**. However, it is also likely that the user would like to view profiles at different levels of granularity (e.g., one profile per **RBAC_ROLE_ID**). Just like OLAP, it is convenient to think of these different granularities as forming a partial order [28].

Of course, even for conventional aggregates (which are typically integers or reals), fully-materializing an entire data cube is space-consuming, and because profile objects are larger than integers or reals, space is an even bigger issue here. To select the best set of profiles, given limited space, we implemented the heuristic proposed by Harinarayan et al. [29] in a separate **Planner** module.

⁶The proof is straightforward, but is omitted for space.

To answer queries involving a particular profile, when the profile cube has been partially materialized, we first check to see whether the desired profile has been materialized. If it has, then it can be used directly. Otherwise, ideally, we would like to compute the desired profile from a set of (materialized) constituent profiles at a finer level of granularity. (For example, if a query requests profiles grouping-by `ROLE_ID`, we would like to be able to compute these profiles directly from the set of profiles grouping-by `USER_ID`, rather than using the original data.) Like standard aggregate data cubes, this is captured by the idea of *distributive* and *algebraic* aggregates; an aggregate function is algebraic if it can be computed from intermediate statistics at finer levels of granularity [28]. Our internal profile representation is algebraic; the internal counter representation can be computed from profiles at finer granularities without accessing the underlying data. More generally, recent work showed that Naive Bayes classifiers and kernel density estimators are algebraic aggregates [19].

4.2 Profile Compression

Materializing profiles can still be space-consuming, particularly if there are many distinct values for certain features. Thus, it is beneficial to consider compressing materialized profiles. One approach is to replace each of the n counter vectors with a histogram. We apply V-Optimal histogram [32] to each dimension to compress the vector of counts representation. The effect of compression is demonstrated in Section 5.3.

Practically-speaking, compression is incorporated into our prototype via a function `compress(Profile)`. We envision applying compression to archived profiles constructed over historical data (e.g., from past years or months) that is no longer being updated.

5. EXPERIMENTAL EVALUATION

We conducted an extensive experimental study using the Splash system, designed to measure the following:

- **Performance and Scale:** One of our guiding principles was to develop a system that scales to large datasets. We measured the performance and scalability of Splash, including the effects of materialization and compression.
- **Query Language and Profile Abstraction:** To provide a greater understanding of the flexibility provided by Splash, we performed a case study comparison. Using a well-known dataset from the network intrusion domain, and an interesting set of ad-hoc exploratory analysis tasks, we compared the process of expressing these tasks in Splash with the process of expressing these tasks in an open-source data mining package called Weka [13].
- **Generating Representatives:** Finally, we evaluated and compared the algorithms described in Section 3 for generating representatives.

We ran our experiments on a machine with Pentium dual-core *2GHz* CPU and *2GB* memory. We used the Ubuntu 8.04 operating system and PostgreSQL 8.3. The size of shared-memory for PostgreSQL was set to *512MB*.

5.1 Performance & Scale

Our first set of experiments were designed to measure the performance and scalability of Splash when used for large input datasets.

5.1.1 Data Generator

For these experiments, we used a simple synthetic data generator, which produces data with the following schema:

```
SynData(yy, mm ,dd, featureVector)
```

`yy`, `mm`, and `dd` are hierarchical dimension attributes representing the date. `featureVector` is a 10-dimensional feature vector. We generated the values of the features using Gaussian distributions; however, since the synthetic data is used only for testing performance and scale, the distribution is of relatively little importance. Each resulting record is approx. 150 bytes.

For each distinct day, we generated 1000 records. (The average profile size per day is *40KB*.) In the experiments, we vary the number of days (months, and years), in order to vary the size of the dataset. The maximum data size is 100,000 days (roughly 100 million records), which consume *15GB* of total space.

5.1.2 Scale-Up

To evaluate how Splash scales to large input datasets, we varied the input data size from 100 days (*15MB* of data) to 100,000 days (*15GB* of data), and we issued the following profile construction query:

```
SELECT profile(featureVector)
FROM SynData
GROUP BY yy,mm,dd
```

The size of the profiles resulting from this query ranges from *4MB* to *4GB*. For the sake of comparison, we also tested the built-in aggregate `sum()` on the same input. The results of this experiment are shown in Figure 3(a-c).

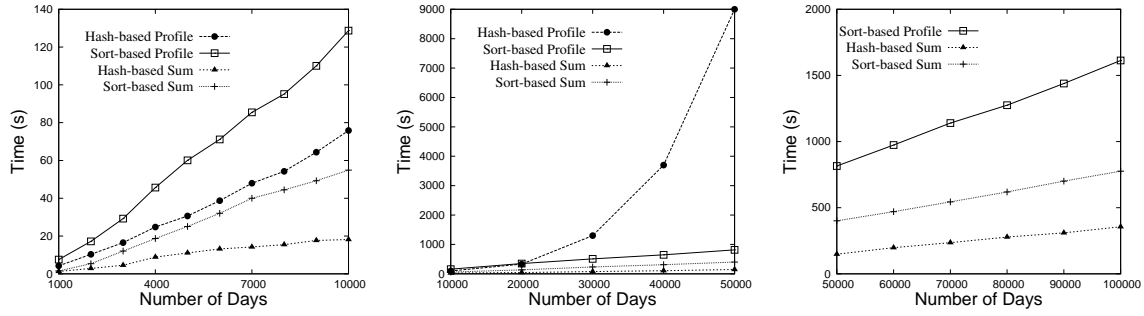
In Figure 3(a), notice that when the data size is small, the `profile()` aggregate works pretty well (using approximately 4 times as much time as `sum()`). However, as the data size grows, the running time for `profile()` suffers significantly. After analyzing the query plan, we discovered that the PostgreSQL query optimizer always selected a hash-based aggregation operation instead of a sort-based plan. However, the total size of the resulting profiles grows much larger than traditional aggregates, and when the hash table containing these profiles can no longer fit in the shared buffer, the query begins to thrash.⁷

On the other hand, if we force the system to use a sort-based plan, we do not encounter this problem. To force PostgreSQL to use a sort-based plan, we explicitly perform the sort as part of the query:

```
SELECT profile(featureVector)
FROM (SELECT * FROM SynData ORDER BY yy,mm,dd)
GROUP BY yy,mm,dd
```

In Figure 3(a), when the data is small, the sorting-based `profile()` is a bit slower than the hash-based `profile()` due to the overhead of sorting. However, as the data size grows, and the hash-based `profile()` begins to thrash (Figure 3(b)), the sort-based operation is not significantly affected. As we keep growing the total size of profiles to *4GB* (Figure 3(c)), which is larger than *512MB* shared-buffer and *2GB* memory, the sort-based `profile()` still scales well.

⁷Interestingly, the current interface for user-defined aggregation in PostgreSQL does not reflect the size of the aggregation results (i.e., a large profile object as opposed to an integer). However, we expect that a simple addition to the interface would provide the optimizer with better information, allowing it to choose hashing vs. sorting.



(a) Total profile size 40M - 400M (b) Total profile size 400M - 2G (c) Total profile size 2G - 4G
Figure 3: Scalability of the aggregate profile()

5.1.3 Effects of Materialization

To demonstrate the effectiveness of materialization, we vary the size of data from 1 year (50M data) to 10 years (500MB data). Suppose that we have constructed (using the synthetic data) a partial materialization including one profile per day, as defined by the following view:

```
CREATE MATERIALIZED VIEW ByDay AS
SELECT yy, mm, dd, profile(featureVector)
FROM SynData
GROUP BY yy, mm, dd
```

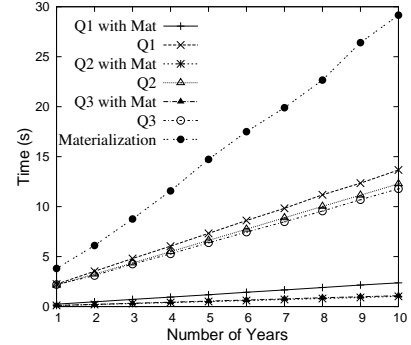
The size of the materialized view varies from 13.3MB (for 1 year) to 133MB (for 10 years). Now, consider the following three queries, each of which can be evaluated either using the materialized table (ByDay) or using the base data (SynData).

- ```
SELECT sim(profile,featureVector)
FROM SynData F,
(SELECT yy,mm,dd, Profile(featureVector)
FROM SynData
GROUP BY yy,mm,dd) AS P(yy,mm,dd,profile)
WHERE P.yy=F.yy AND P.mm=F.mm AND P.dd=F.dd
```
- ```
SELECT sim(profile,featureVector)
FROM SynData F,
(SELECT yy,mm, Profile(featureVector)
FROM SynData
GROUP BY yy,mm) AS P(yy,mm,profile)
WHERE P.yy=F.yy AND P.mm=F.mm
```
- ```
SELECT sim(profile,featureVector)
FROM SynData F,
(SELECT yy Profile(featureVector)
FROM SynData
GROUP BY yy) AS P(yy,profile)
WHERE P.yy=F.yy
```

Figure 4 shows the performance of each query using the materialized table (“Mat”) and using the base data. The running time drops substantially when the materialized table is used. For the two queries that use profiles grouping by month and by year, although no materialized table can be used directly (i.e., ByMonth or ByYear), calculating these profiles from the materialized ByDay is still more efficient than building them from scratch.

## 5.2 Application Case Study

Our next set of experiments evaluates the flexibility afforded by a high-level query language incorporating profiles. For this purpose, we conducted a case study for network attack logs. The idea is to consider a security administrator and the ad-hoc exploratory tasks he or she would conduct



**Figure 4: Effect of materialization**

to understand the logged information. We chose this particular domain because anomaly-based intrusion detection has been well-studied for networks, and there exist established benchmark data sets.

Existing data mining and statistical software packages (e.g., Weka [13], Matlab [7], and R [9]) do not provide flexible high-level query languages. This means that if a user wants to make advanced use of statistical or mining primitives (e.g., embedding in a larger analysis), then she must write a custom procedural program each time. Further, query optimization strategies cannot be automatically incorporated into these programs, so the user also needs to worry about optimization each time she writes such a script.

For our case study, we developed a sequence of exploratory tasks, and we compared the programming effort required to express these tasks using Splash and using Weka<sup>8</sup> [13]. We observed that it is relatively easy to express conventional tasks (e.g., simple classification) in both systems, primarily because Weka provides a custom API for these tasks. However, when the analysis task involves additional data processing, or compound tasks, it is necessary to embed calls to the Weka API in a larger (custom-coded) program, which is inconvenient and time-consuming for ad-hoc analysis. In contrast, compound tasks can usually be expressed quite easily in Splash.

### 5.2.1 Network Attack Data

For the case study, we used the KDD Cup 1999 dataset[6], which has been heavily studied, and is considered a benchmark for data mining techniques and intrusion detection. The data set consists of a set of Internet connection records

<sup>8</sup>Of course, there are other software packages (e.g., [39], [7], [9]). We selected Weka because it is popular, and representative of software packages that do not support declarative languages.

(a connection is a sequence of TCP packets), each with 41 pre-extracted features (including, for example, the protocol type of the connection, network service type of the destination, and number of data bytes transferred from source to destination).

The connections are divided into five main classes: normal connections (*normal*), probe attacks (*probe*), denial-of-service attacks (*dos*), user-to-root attacks (*u2r*), and remote-to-local attacks (*r2l*). In addition, the four main attack classes can be further decomposed into sub-classes. For example, the *dos* attack is decomposed into sub-classes that include *dos.apache2*, *dos.mailbomb*, and *dos.updstorm*. The data set consists of 494,021 training records, and 311,029 testing records. We store these data sets (including extracted features) in two tables, where `id` uniquely identifies each record, `featureVector` stores the 41 features associated with the connection, `class` stores the main class label associated with the connection, and `subClass` stores the sub-class:

```
TrainingData(id, featureVector, class, subClass)
TestingData(id, featureVector, class, subClass)
```

The KDD Cup contest posed a classification task: Using the training data, construct a model that accurately predicts the attack category (among the 5 main classes) for each of the testing connections. Classification results were evaluated by taking the average error across all classification decisions (on the testing set), so a lower score implies better results. The best reported score in the contest was 0.2331, and results within the range [0.2331, 0.2684] were considered good [10].

### 5.2.2 Case Study Tasks and Results

We performed a series of exploratory tasks (including, but not limited to, simple classification) to help understand the information contained in these logs. For each task, we compared the process of performing the task using Splash with the process of performing the task in Weka [13].

For this comparison, we found a meaningful quantitative measure elusive. When we began the study, we started by comparing the number of lines of code necessary to perform each task in Splash vs. Weka. However, we found that this did not fully convey the distinctions between the two systems. (As one example, some of the tasks could be accomplished in Weka by modifying the engine with a small amount of code, or by writing a larger amount of application code, and it was not clear how to quantify the difference between these two solutions.) Thus, we deliberately chose to provide a qualitative comparison, rather than quantitative measurements.

#### 1. Classification

**Task:** Classification is a common data mining technique that can be used in network intrusion detection when training data is available describing both normal behavior and attacks [43].

**Splash:** For simplicity, suppose that we begin by constructing one profile for each of the 5 main class labels:

```
CREATE VIEW Profiles(class, count, profile)
AS SELECT class, count(*),profile(featureVector)
FROM TrainingData
GROUP BY class
```

Classification of the testing records is performed by comparing each record with all of the profiles to find the profile

with maximum similarity. (For the sample profile construction and similarity functions described in Sections 2.3, the following trains a Naive Bayes classifier, and uses it to classify the testing records.)

```
CREATE VIEW MaxSim(id, maxsim) AS
SELECT T.id, max(sim(P.profile, T.featureVector)*P.count)
FROM TestingData T, Profiles P
GROUP BY T.id
```

```
SELECT T.id, P.class
FROM TestingData T, Profiles P, MaxSim M
WHERE T.id = M.id
AND sim(P.profile, T.featureVector)*P.count = M.maxsim
```

We observe that this simple classifier produces a score of 0.258 for the KDD Cup Dataset, which is considered good [10]. Of course, our goal is *not* to test the classification algorithm itself (Naive Bayes), but this serves as a sanity check, indicating that our sample model is reasonable for the experiments.

**Weka:** Classification is one of the standard data mining tasks for which Weka provides an explicit API. It is useful to note that training and testing a classifier (e.g., a Naive Bayes model) uses around ten lines of code, which is comparable to the size of the query required by Splash.

#### 2. Anomaly Detection

**Task:** In contrast to classification, anomaly detection is practical when the only available training data describes normal behavior, rather than malicious or attack behavior. Intuitively, the goal is to construct a model describing normal behavior, and then mark those records that are dissimilar to the model as potential attacks [43].

**Splash:** We can express anomaly detection in Splash as follows, where `thres1` is a parameter provided by the security administrator:

```
SELECT T.fid
FROM TestingData T, Profiles P
WHERE P.class = 'normal'
AND sim(P.profile, T.featureVector) < thres1
```

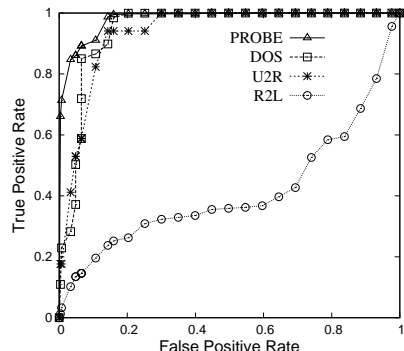
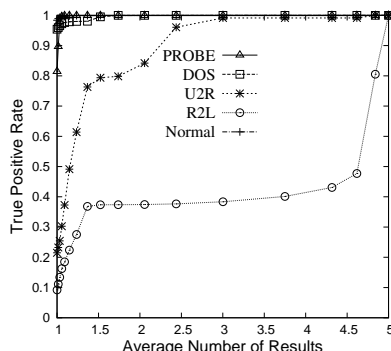
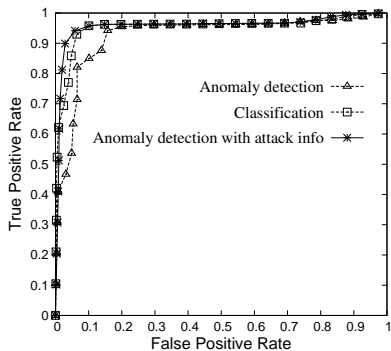
**Weka:** Weka does not provide an explicit API for anomaly detection, which leaves the user with two options: (1) She can modify the Weka engine (altering the API) in order to add such functionality. (In doing this, she can re-use some of the existing engine-level classification code.) (2) She can write a custom application from scratch to implement anomaly detection.

#### 3. Anomaly Detection Leveraging Attack Data

**Task:** We observed that classification and anomaly detection each have strengths and weaknesses. In particular, classification cannot identify unknown attacks that do not appear in the training data, but anomaly detection cannot leverage training data for known attacks [24]. A logical new task exploits both tools: An audit record is considered part of an attack if it is classified as an attack by a classifier, *or* it is marked as anomalous by an anomaly detector. (Related ideas appear in [37, 48, 52].)

**Splash:** This task combines anomaly detection and classification, and is expressed using the following simple query:

```
SELECT T.id
FROM TestingData T, Profiles P, MaxSim M
WHERE P.class = 'normal'
AND (sim(P.profile,T.featureVector) < thres1
OR sim(P.profile,T.featureVector)*P.count != maxsim)
```



**Figure 5: Leveraging attack info. in Figure 6: Tuning recall for classification** **Figure 7: Tuning false positive rate for anomaly det.**

The effects of integrating anomaly detection and classification are shown in Figure 5. While our goal is not to develop new machine learning tools, it is interesting to observe that the composite analysis outperforms both anomaly detection and classification, which points to the importance of supporting a flexible infrastructure.

**Weka:** While Splash provides a convenient set of declarative query operators, in Weka we need to write custom application code encapsulating the basic data mining operations. In this case, the custom application consists of four steps (provided that we have already written a new anomaly-detection module for Weka): (1) Train the classifier; (2) Classify the testing data, and filter those records with class labels other than “normal”; (3) Train the anomaly detector; (4) Identify the anomalies in the remaining testing data. This process consists of two training and two testing phases, which are redundant compared to the Splash query, which constructs a single set of profiles. This redundancy is rooted in the rigid API design of Weka, which does not capture the underlying relationship between statistical anomaly detection and statistical classification. Further, it is the security administrator’s responsibility to take care of interleaving the four phases to make the code efficient.

#### 4. Classification Using a Class Taxonomy

**Task:** We also observed that the KDD cup data actually includes class labels that are expressed at multiple levels of granularity. While each of the main categories (i.e., *dos*, etc.) is present in the training data, the testing data contains some new sub-categories that are not present in the training data. Thus, we considered the classification problem of assigning the best class label (at any granularity) to each testing record.

**Splash:** Splash is particularly well-suited to handle this task. The following constructs a view which contains profiles for all class and sub-classes in the training data:

```
CREATE VIEW Profiles(class, count,profile) AS
SELECT class, count(*),profile(featurevector)
FROM TrainingData
GROUP By class
UNION
SELECT subClass, count(*),profile(featurevector)
FROM TrainingData
GROUP BY subClass
```

Using this view, the same query as in the classification task can be used to classify each testing record to the most appropriate class (at any level of granularity). In order to compare this to the original result, we mapped the results

back to the 5 main classes. Our goal, of course, is to demonstrate that it is simple to express even complex tasks using Splash. However, it is interesting to observe that this approach attains a score of 0.228 (better than any result reported as part of the KDD Cup contest!), which points to the importance of supporting complex tasks.

**Weka:** We see two ways of approaching this task using Weka. The first option would extend one of Weka’s classifiers (e.g., Naive Bayes) to make the classifier aware of the hierarchical class labels present in the training data. This requires modifications to the Weka engine and API.

The second option is done entirely at the application level. Notice that the hierarchical relationships between class labels can be handled without modifying the classifier by producing additional training data. For example, if there exists a training record with label *attack.dos.apache2*, we can generate three training records with labels *attack*, *dos*, and *apache2*. However, this approach unnecessarily expands the size of the training data by a factor of three.

#### 5. Tuning Classification Recall

**Task:** While classification and anomaly detection tools are powerful, they do not provide the built-in ability to tune the results according to a security administrator’s specification. For example, in the KDD Cup contest, even the winning solution observed very low true positive rates for detecting *u2r* and *r2l* attacks (13.2% and 8.4%, respectively). Thus, when conducting an exploratory analysis, the security administrator might consider examining more than one class label per request, as a way of boosting the number of true positives.

**Splash:** Using Splash, the classification recall can be adjusted simply by modifying *thres2* in the following query:

```
SELECT T.id, P.class
FROM TestingData T, Profiles P, Maxsim M
WHERE T.id = M.id
AND sim(P.profile,T.featureVector)*P.count
>=M.maxsim-thres2
```

The effects of tuning *thres2* are shown in Figure 6. Notice that by returning an average of 1.4 class labels per testing record, the security administrator can achieve 79.0% and 37.4% recall for the *u2r* and *r2l* attacks, which are 3.3 and 4.2 times better, respectively, than the recall rate when returning just one class label per test record.

**Weka:** The problem of tuning classification recall should be easy to handle in Weka. However, the Weka API is not designed to accept the parameter *thresh2*, or to return mul-

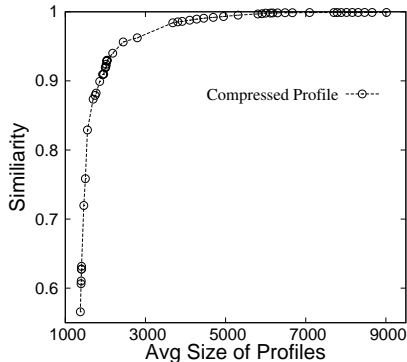


Figure 8: Effect of compression on similarity to original profiles

tuple class labels. Thus, to evaluate this task would require some re-design and modifications to Weka.

## 6. Tuning False Positives for Anomaly Detection

**Task:** Finally, high false positive rates are a major concern when adopting anomaly detection techniques [43], and the security administrator may want to compromise some true positives for a lower false positive rate. In either case, the ability to tune the result set is critical for exploratory analysis.

**Splash:** False positives are easily tuned using parameter *thres1* in the above anomaly detection query. Figure 7 shows the tradeoff between true positives and false positives.

Of course, finding an appropriate *thres1* is important in anomaly detection. Given a set of training data, and the maximum allowable false positive rate  $r$ , we can estimate the parameter *thres1* from the training data. In the following, let  $n = r \cdot |TrainingData|$ .

```
SELECT sim(P.profile,T.featureVector) as thres1
FROM TrainingData T, Profiles P
WHERE T.class = 'normal' AND P.class = 'normal'
ORDER BY sim(P.profile,T.featureVector)
LIMIT 1 OFFSET n
```

This query orders the training records based on their similarity to the profile of normal behavior, finds the  $n^{th}$  most similar training record, and uses its similarity to the normal profile to set the threshold.

**Weka:** Weka does not currently support anomaly detection, but if we were to add this functionality, we would need to incorporate *thres1* as a parameter in the API in order to support this task.

## 5.3 Effects of Compression

While compression based on histograms, as described in Section 4.2, is effective in reducing the size of profile objects, it also removes some detail from the profiles. To measure the effects of compression, we performed several experiments using the KDD Cup data. In particular, we used four metrics to evaluate the loss of content resulting from histogram-based compression: (1) Similarity between the compressed profile and the original profile, (2) Accuracy of a classifier constructed using the compressed profiles, (3) True positive rate for anomaly detection, and (4) False positive rate for anomaly detection. Figure 8 plots the similarity between the compressed and original profiles. The other results are omitted for space; however, we can reduce the profiles to 25% of their original size without substantially altering any of the four evaluation metrics.

## 5.4 Generating Representatives

Our final set of experiments evaluated and compared algorithms for finding representative sets. We tested the three algorithms described in Section 3 – HIST-R, RAND-R, and GREEDY-R – as well as an algorithm that selects the  $k$  maximum likelihood records. This last approach is equivalent to the notion of *typical tuples* proposed in [31]; this work was driven by a different problem formulation, and as will show, the resulting algorithms do not suit our purposes.

The first experiments use an input data set  $D$  containing 10,000 records. Profile  $p$  is taken to be an  $n$ -dimensional Gaussian distribution, and  $D$  is generated according to the same distribution. The results for  $d = 1, 2, 5$  are shown in Figure 9. GREEDY-R produces the best results; in addition, we note that although HIST-R is more effective than RAND-R in low dimensions, as the dimensionality increases, HIST-R is not better than RAND-R. The maximum likelihood method is clearly not suitable for this problem. We also observed similar results for other distributions (zipf and uniform), but the results are omitted for space.

We also test the case where  $D$  comes from a distribution other than profile distribution  $p$ . Figure 9(d) shows the case where  $D$  comes from a 5-dimensional Zipf distribution, but  $p$  is a 5-dimensional uniform distribution; Figure 9(e) shows the case where  $D$  comes from a Gaussian distribution ( $\sigma = 20$ ) and  $p$  is Gaussian ( $\sigma = 10$ ). It is interesting to observe that using GREEDY-R it is still possible to select from  $D$  a limited number of tuples that represent a different distribution  $p$ .

Finally, we compared the algorithms using five features extracted from the KDD Cup data. The results (Figure 9(f)) are as expected; GREEDY-R works the best, but due to the dimensionality, HIST-R and RAND-R do not perform as well.

## 6. RELATED WORK

**Integrating Databases and Statistical Models:** Splash is built on the idea of simultaneously querying statistical models and data. A body of related literature describes several other abstractions and systems for incorporating data mining and statistical models into standard or extended SQL. Example systems include MauveDB [22], Microsoft SQL Server’s DMX [2] and OleDB for DM [40], and IBM DB2 Intelligent Miner [3]. However, for reasons outlined in Section 1.2, none of these systems satisfied our requirements, particularly ad-hoc model creation and manipulation.

Like Splash, the ATLaS system [50] used user-defined aggregates to implement simple data mining operations (e.g., association rules and decision trees). While the idea is similar, ATLaS supported a lower-level language, which required the user to code the details of the mining algorithm. In this way, Splash and ATLaS complementary; an expert codes the statistical inference algorithms using the user-defined aggregate interface of Postgres (which is similar to that supported by ATLaS), and then Splash provides a simpler high-level query language for less-sophisticated users.

A significant amount of work has focused on implementing specific data mining operations inside relational databases [41, 45, 46], often obtaining good performance and scale. There is also a lot of work trying to design scalable data mining algorithms [16, 27]. However, none of these works have focused on integration with ad-hoc query languages.

Recent work MAD [20] focused on designing statistical algorithms feasible for running on parallel databases. How-

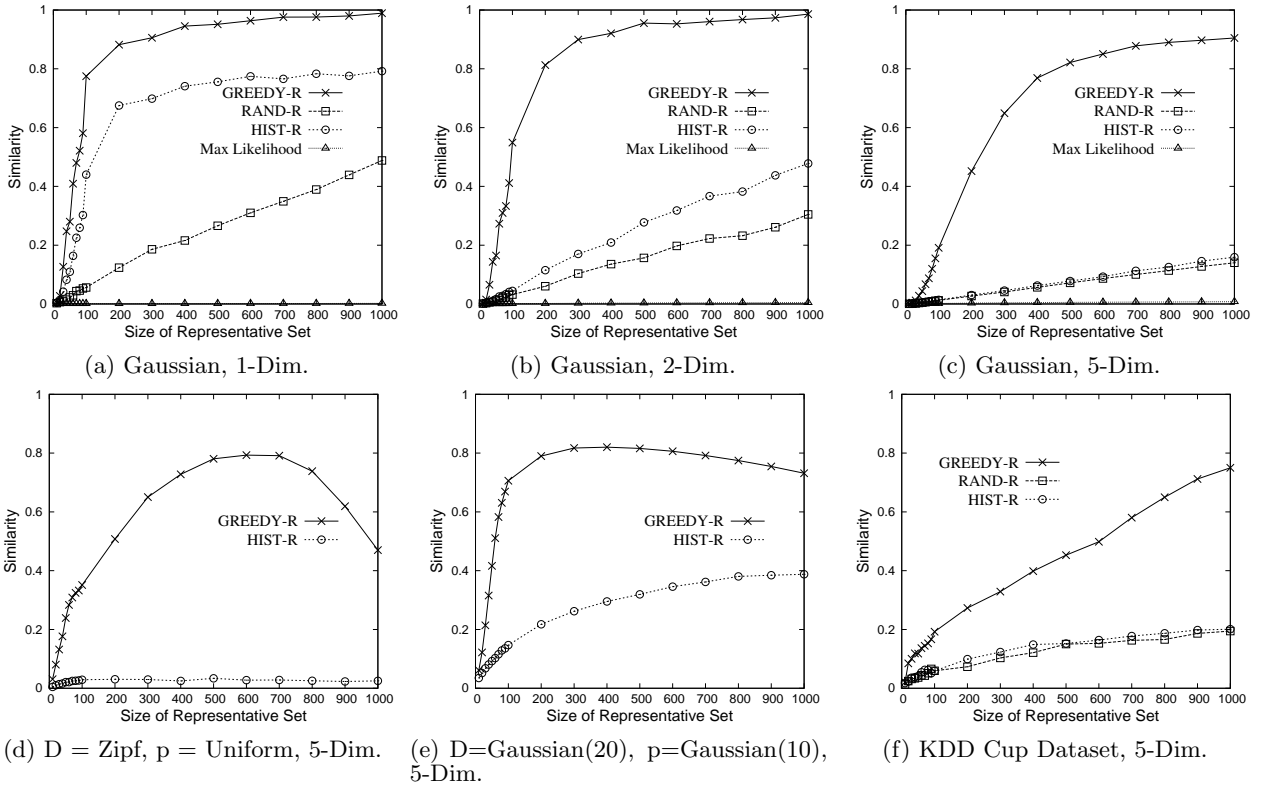


Figure 9: Finding representative set from data set of various dimension

ever, the paper does not provide abstraction for *statistical model* objects. Therefore it is difficult to write queries to compare and analyze models.

**Exploratory Data Mining:** Recent work in exploratory data mining has proposed to view data mining tasks in terms of cube space. For example, Chen et al. proposed the idea of a prediction cube, where classifiers are trained over various data subsets in multidimensional cube space [19]. While this approach is related to our view of statistical models as aggregate functions, to the best of our knowledge, none of the past work has considered integrating cube-based data mining with a relational DBMS or query language.

**Representative and Typical Tuples:** As part of our system, we also studied the problem of generating “representative” examples to help explain a profile (Section 3).

To the best of our knowledge, the problem of explaining a statistical model using a small number of examples has not previously been considered. However, related work has considered various formulations of the following problem: Given a dataset  $D$ , produce a “representative” subset thereof, where “representative” is defined in different ways.

Hua et al. considered finding the top- $k$  most “typical” (maximum likelihood) tuples [31]; however, as we showed in Section 5.4, the typical tuples often fail our goal of clearly describing the underlying distribution. Liu and Jagadish present a problem formulation based on distance; that is, minimize the distance between the points in  $D$  and their representatives [38]. While this approach captures the diversity of data records, it also does not convey the frequency distribution in  $D$ . Finally, Pan et al. [42] developed an objective function based on information theory, but it has a different goal from ours in finding a subset from a transactional (itemset) database that simultaneously has high coverage and low

redundancy.

**Anomaly and Fraud Detection:** One of our motivating applications is ad-hoc log analysis. The idea of anomaly-based intrusion detection goes back to the work of Denning [21]. Typically, however, anomaly detection is viewed as a binary decision (i.e., produce a warning or not), and false positives are often cited as a shortcoming. This provides strong motivation for an ad-hoc query tool in this domain.

Anomaly detection has been studied extensively in operating systems [26, 35, 36, 51], networks (e.g., see recent survey [43]), and for detecting fraud [17, 25]. It is distinct from signature-based intrusion detection (e.g., [4]) which detects pre-defined patterns of abnormal behavior.

Recent work has begun to consider applying anomaly detection to databases. Kamra et al. [33] developed a simple data mining approach, which selects features from the text of SQL queries, and constructs a Naive Bayes classifier to predict the most likely profile (in this case, the RBAC role) for new queries. Other work includes [30, 34, 47, 49].

## 7. CONCLUSION & FUTURE WORK

In this paper, we presented *Splash*, a novel system supporting ad-hoc querying of statistical models and relational data. The fundamental new abstractions supported by *Splash* are the view of statistical models as SQL aggregation operations, as well as operations for interacting with models, including the generation of representative examples. We provided an implementation of *Splash*, including several performance optimizations. Further, an extensive experimental study indicates both that the system scales well, and that the novel abstractions provide a simple alternative to the existing (more complex and rigid) APIs supported by data mining software packages.

In the future, we would like to run *Splash* in a parallel

database. Standard SQL aggregation functions have been successfully implemented in parallel databases for many years [23], and by analogy, we believe the abstractions supported by Splash lend themselves to easy parallelization of certain mining tasks.

## Acknowledgements

We thank H. V. Jagadish for his invaluable comment on an earlier version of this paper. This work was supported in part by NSF grant CNS-0915782 and IIS-0438909.

## 8. REFERENCES

- [1] <http://www.hhs.gov/ocr/hipaa/>.
- [2] Data mining extensions (DMX) reference. SQL Server 2005 Books Online. <http://technet.microsoft.com>.
- [3] Db2 intelligent miner. <http://www-01.ibm.com/software/data/iminer/>.
- [4] <http://www.snort.org>. Retrieved July 16, 2008.
- [5] An introduction to computer security: The NIST handbook. NIST Special Publication 800-12.
- [6] Kdd cup 1999 dataset. <http://archive.ics.uci.edu/ml/databases/kddcup99/kddcup99.html>.
- [7] Matlab: The language of technical computing. <http://www.mathworks.com/products/matlab/>.
- [8] Postgresql. <http://www.postgresql.org/>.
- [9] The r project for statistical computing. <http://www.r-project.org/>.
- [10] Result of kdd cup 1999 contest. <http://www-cse.ucsd.edu/~elkan/clresults.html>.
- [11] Sas: Business intelligence software. <http://www.sas.com>.
- [12] Stata. <http://www.stata.com>.
- [13] Weka 3: Data mining software in java. <http://www.cs.waikato.ac.nz/ml/weka/>.
- [14] University of Michigan Health System Compliance Office. Personal communication, 2008.
- [15] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.
- [16] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.
- [17] F. Bonchi, F. Giannotti, G. Mainetto, and D. Pedreschi. A classification-based methodology for planning audit strategies in fraud detection. In *SIGKDD*, 1999.
- [18] J. Cart. Kaiser fires staffers who snooped into suleman's files. *The Los Angeles Times*, March 31 2009.
- [19] B. Chen, L. Chen, Y. Lin, and R. Ramakrishnan. Prediction cubes. In *VLDB*, 2005.
- [20] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. Mad skills: New analysis practices for big data. In *VLDB*, 2009.
- [21] D. Denning. An intrusion-detection model. In *IEEE Symposium on Security and Privacy*, 1986.
- [22] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.
- [23] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.
- [24] P. Dokas, L. Ertöz, V. Kumar, A. Lazarevic, J. Srivastava, and P. Tan. Data mining for network intrusion detection. In *Proceedings of NSF Workshop on Next Generation Data Mining*, 2002.
- [25] T. Fawcett and F. Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1, 1997.
- [26] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, 1996.
- [27] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large dataset. In *VLDB*, 1998.
- [28] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1996.
- [29] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cube efficiently. In *SIGMOD*, 1996.
- [30] Y. Hu and B. Panda. Identification of malicious transactions in database systems. In *IDEAS*, 2003.
- [31] M. Hua, J. Pei, A. Fu, X. Lin, and H. Leung. Efficiently answering top-k typicality queries. In *VLDB*, 2007.
- [32] Y. Ioannidis and Y. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, 1995.
- [33] A. Kamra, E. Terzi, and E. Bertino. Detecting anomalous access patterns in relational databases. *VLDB Journal*, 2007.
- [34] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *CCS*, 2003.
- [35] W. Lee and S. Stolfo. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Methods in Fraud and Risk Management*, 1997.
- [36] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *USENIX Security Symposium*, 1998.
- [37] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, 1999.
- [38] B. Liu and H. Jagadish. Using trees to depict a forest. In *VLDB*, 2009.
- [39] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *SIGKDD*, 2006.
- [40] A. Netz, S. Chaudhuri, U. Fayyad, and J. Bernhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *ICDE*, 2001.
- [41] C. Ordonez. Building statistical models and scoring with udfs. In *SIGMOD*, 2007.
- [42] F. Pan, W. Wang, A. Tung, and J. Yang. Finding representative set from massive data. In *ICDM*, 2005.
- [43] A. Patcha and J. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51:3448–3470, 2007.
- [44] Y. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.
- [45] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.
- [46] K. Sattler and O. Duneman. Sql database primitives for decision tree classifiers. In *CIKM*, 2001.
- [47] A. Spalka and J. Lehnhardt. A comprehensive approach to anomaly detection in relational databases. In *Proceedings of the 19th IFIP WG 11.3 Working Conference on Data and Applications Security*, 2005.
- [48] E. Tombini, H. Debar, L. Me, and M. Ducasse. A serial combination of anomaly and misuse idses applied to http traffic. In *ACSAC*, 2004.
- [49] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of SQL attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2003.
- [50] H. Wang and C. Zaniolo. Atlas: A native extension of sql for data mining. In *SIAM Data Mining*, 2003.
- [51] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.
- [52] J. Zhang and M. Zulkernine. A hybrid network intrusion detection technique using random forests. In *Conference on Availability, Reliability and Security*, 2006.
- [53] Y. Zhang, H. Herodotou, and J. Yang. Riot: I/o-efficient numerical computing without sql. In *CIDR*, 2009.