Hidden Database Research and Analytics (HYDRA) System

Yachao Lu[†], Saravanan Thirumuruganathan[‡], Nan Zhang[†], Gautam Das[‡]
The George Washington University[†], University of Texas at Arlington[‡]
{yachao, nzhang10}@gwu.edu[†], {saravanan.thirumuruganathan@mavs, gdas@cse}.uta.edu[‡]

Abstract

A significant portion of data on the web is available on private or hidden databases that lie behind form-like query interfaces that allow users to browse these databases in a controlled manner. In this paper, we describe System HYDRA that enables fast sampling and data analytics over a hidden web database with a form-like web search interface. Broadly, it consists of three major components: (1) SAMPLE-GEN which produces samples according to a given sampling distribution (2) SAMPLE-EVAL that evaluates samples produced by SAMPLE-GEN and also generates estimations for a given aggregate query and (3) TIMBR that enables fast and easy construction of a wrapper that models both input and output interface of the web database thereby translating supported search queries to HTTP requests and retrieving top-k query answers from HTTP responses.

1 Introduction

1.1 Motivation

A large portion of data available on the web is present in the so called "Deep Web". The deep web consists of private or hidden databases that lie behind form-like query interfaces that allow users to browse these databases in a controlled manner. This is typically done by search queries that specify desired (ranges of) attribute values of the sought-after tuple(s), and the system responds by returning a few (e.g., top-k) tuples that satisfy the selection conditions, sorted by a suitable ranking function. There are numerous examples of such databases, ranging from databases of government agencies (such as data.gov, cdc.gov), databases that arise in scientific and health domains (such as Pubmed, RxList), to databases that occur in the commercial world (such as Amazon, EBay).

While hidden database interfaces are normally designed to allow users to execute search queries, for certain applications it is also useful to perform *data analytics* over such databases. The goal of data analytics is to reveal insights and "big picture" information about the underlying data. In particular, we are interested in data analytics techniques that can be performed only using the public interfaces of the databases while respecting the data access limitations (e.g., query rate limits) imposed by the data owners. Such techniques can be useful in powering a multitude of third-party applications that can effectively function anonymously without having to enter into complex (and costly) data sharing agreements with the data providers. For example, an economist would be interested in tracking economically important aggregates such as the count of employment postings in various categories every month in job search websites.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In the literature, the three common data analytics tasks have been *crawling*, *sampling*, and *aggregates estimation*. The objective of crawling is to retrieve all tuples from the database. The tuples thus collected can serve as a local repository over which any analytics operation can be performed. Unlike crawling, sampling aims to collect only a small subset of tuples of the database drawn randomly according to a pre-determined sampling distribution. This distribution can be uniform (leading to a simple random sample) or can be weighted. If collected appropriately, the sample is representative of the database and can therefore be used for a wide variety of analytics purposes. Aggregate estimation is the final task which aims to answer aggregate queries (AVG, SUM, COUNT, etc.) over the database, so as to enable data analytics and mining applications that can be built upon the answered aggregates. Aggregate estimation is better suited when we already know which aggregate needs to be estimated (thereby allowing us to tailor the analytics process for the specific aggregate in question), while sampling is more flexible but may not be as efficient for estimating a given aggregate. All of these tasks may be carried out over a single snapshot of a hidden database or continuously run when the database changes over time.

Today, data analytics over hidden databases face the following significant challenges:

- **No direct way of executing data analytics queries:** The input search interface of a hidden database is usually a web form, allowing a user to only formulate simple queries, such as conjunctive queries or kNN nearest neighbor queries. There is no direct way of specifying aggregate or data analytics queries. Likewise, the returned results are revealed via an output interface that limits the number of tuples returned (the *top-k constraint*) ordered by an often-proprietary ranking function. This makes data analytics problems such as sampling difficult, as the returned tuples for a query are not necessarily representative samples of the underlying database, e.g., there may be a bias towards tuples "favored" by the ranking function used by the website - which for a e-commerce website could mean products with lower prices, higher customer ratings, etc.

- **Query rate Limitations:** Most hidden databases limit how many search queries (essentially HTTP requests) a user (or an IP address, an API account, etc.) can issue over a given time period. Search APIs are limited to 5000 queries per day in EBay or 180 queries every 15 minutes in Twitter. Since search queries form the only access channel we have over a hidden database, this limit requires us to somehow translate any data analytics task into a small number of search queries, feasible to issue within a short time period even under the query rate limitation. The bad news is that the query rate requirement for the data analytics task of *crawling* is too large to be practical under the query rate limitation enforced by real-world hidden databases.Therefore, crawling is not a focus of this paper, and we henceforth only consider sampling and aggregate estimation.

- **Understanding the query interface:** Thus far in our discussions, there is an implicit assumption that the query interface of a hidden database has already been "understood", i.e., a site-specific wrapper has already been designed to enable mapping of user specified queries into the web form and to retrieve tuples from the returned result page. In reality, this is an extremely difficult task to automate for arbitrary websites as well to maintain the wrappers when the interface designs are updated by the websites. Understanding query interfaces and wrapper generation are well-recognized research problems in the area of information extraction.

1.2 System Hydra

We develop System Hydra (Hidden Database Research and Analytics) which addresses the above challenges of data analytics over hidden databases. It consists of three components: two main components SAMPLE-GEN and SAMPLE-EVAL and one auxiliary component TIMBR. SAMPLE-GEN enables the user to specify a sampling distribution and then obtain samples according to the specified distribution. SAMPLE-EVAL, on the other hand enables the estimation of a user-specified aggregate query. If the hidden database of concern provides a standardized search query interface, then the TIMBR component is not really needed. Otherwise, if only a form-like web interface is provided by the hidden database, the user can use the TIMBR component to easily build a wrapper that translates the search queries required by the two main components to HTTP requests and responses to and from the hidden database server.

The main technical contributions of the system can be summarized as follows:

- For the generation of samples in SAMPLE-GEN, our focus is to enable the generation of samples with statistical guarantees while minimizing the number of HTTP requests made to the target hidden database. The motivation here is to avoid overloading the target server, and also because many real-world hidden databases limit the number of requests that can be made from an IP address or user/API account for a given time period. The main techniques in SAMPLE-GEN include two parts: One handles traditional SQL-like query semantics - i.e., the tuples returned must match the selection conditions specified in the input query. The other handles kNN query semantics - i.e., while the returned tuples may not match all selection conditions, they are ordered according to a pre-defined, often proprietary, distance function with the specified conditions. We found that not only these two query-return semantics call for the design of different sampling techniques, there are many other subtler interface issues that affect sampling design - e.g., when the returned results are truncated to the top-k tuples, whether the real COUNT of matching tuples (for the first SQL-like semantics) is also returned. We shall discuss these subtle issues and implications in detail in the paper.

- For the estimation of aggregates in SAMPLE-EVAL, the main focus is on understanding how the generation of (certain) samples affect the end goal, which is often to estimate one or more aggregate queries. The design of SAMPLE-EVAL considers three key factors associated with the usage of a sample tuple for the purpose of aggregate estimations: (1) bias, i.e., whether the expected value of an aggregate estimation (from samples) agrees with its real value; (2) variance, i.e., how adjusting the sampling distribution affects the variance of estimated aggregates, which is as important as bias given that the mean square error of an estimation is the SUM of bias^2 and variance; and (3) longevity, i.e., when the underlying database changes, whether a sample tuple needs to be frequently updated, or remains useful for a long time. We shall discuss in the paper how SAMPLE-EVAL evaluates the three facets and determines how to adjust the sampling process based on the intended application (e.g., aggregates to be estimated) and the information learned so far from the hidden database.

- For understanding the input interface, TIMBR provides two methods: (1) a user can click on any input control (e.g., text-box, dropdown menu) and map it to an attribute of interest, and (2) for websites with more complex designs (e.g., calendar inputs which are not standard HTML components but manually implemented using HTML, CSS and JavaScript), TIMBR allows a user to train the system by recording a sequence of interactions with the web page and mapping the sequence to an input value. For understanding the output interface, TIMBR once again provides two methods to accommodate the different design complexities of webpages for real-world hidden databases. For most websites, the human interaction is as simple as clicking on a tuple of interest and then click on the attributes of interest within the tuple. The TIMBR component can then automatically find other tuples and their corresponding values on the attributes of interest. There are a small number of websites, however, that call for a more subtle design. For example, some websites mix query answers with promotional results (e.g., tuples that do not match the input query). To properly define the results of interest for these websites, TIMBR also provides a method for a human user to specify not one, but two tuples of interest. Our system then automatically learn the differentiator between the tuples of interest and (potentially) other tuples displayed on the web page, enabling the precise selection of returned tuples.

It is important to understand that our goal here is not to develop new research results for the vast field of information extraction, but to devise a simple solution that fits our goal of the HYDRA system - i.e., a tool that a data analyst with substantial knowledge of the underlying data (as otherwise he/she would not be able to specify the aggregate queries anyway) can quickly deploy over a form-like hidden database.

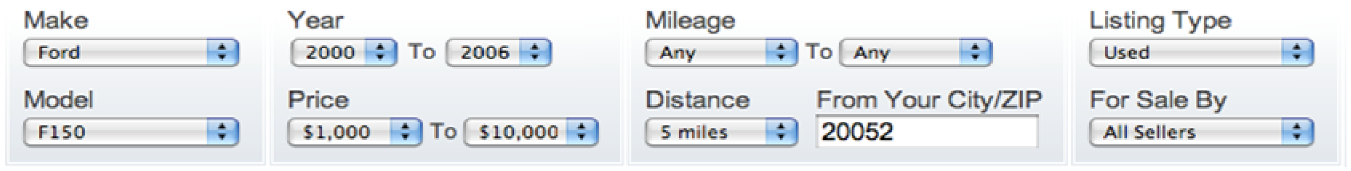
The rest of this paper is organized as follows. Section 2 introduces the data model and a taxonomy of data analytics over hidden databases. Section 3 provides an overview of the system HYDRA and its various components that are described in following sections. Section 4 describes key techniques used by the first component - SAMPLE-GEN - to retrieve samples from underlying hidden web databases. Section 5 focusses on component SAMPLE-EVAL that evaluates the samples collected by SAMPLE-GEN. Section 6 describes the TIMBR component for modeling the input and output interface of a hidden web databases. We describe the related work in Section 7 followed by final remarks and future work in Section 8.

2 Preliminaries

2.1 Model of Hidden Databases

Consider a hidden web database D with n tuples and m attributes $A_1, \dots, A_m$, with the domain of attribute A_i being U_i, and the value of A_i for tuple t being $t[A_i]$. Reflecting the setup of most real-world hidden databases, we restrict our attention to categorical (or ordinal) attributes and assume appropriate discretization of numeric attributes (e.g., price into ranges such as (0, 50], (50, 100], etc., as in hidden databases such as amazon.com).

Input Search Interface: The search interface of a hidden database is usually formulated like a web form (see Figure 1 for an example), allowing a user to specify the desired values on a subset of attributes, i.e., $A_{i_1} = v_{i_1}$ & ... & $A_{i_s} = v_{i_s}$, where $i_1, \dots, i_s \in [1, m]$ and $v_{i_j} \in U_{i_j}$. In Figure 1, the user specifies a query to return cars with predicates `Make=Ford & Model=F150 & ... & ForSaleBy=All Sellers`. While some hidden databases allow any arbitrary subset of attributes to be specified, many others place additional constraints on which subsets can or cannot be specified. For example, most hidden databases require at least one attribute value to be specified, essentially barring queries like `SELECT * FROM` D. Some even designate a subset of attributes to be "required fields" in search (e.g., departure city, arrival city and departure date in a flight search database).



Make: Ford
Model: F150
Year: 2000 To 2006
Price: $1,000 To $10,000
Mileage: Any To Any
Distance: 5 miles
From Your City/ZIP: 20052
Listing Type: Used
For Sale By: All Sellers

Figure 1: A form-like input interface for a Hidden Database

Another, somewhat subtler, constraint is what we refer to as the "auto-filling searches" - i.e., the hidden database might use information specified in a user's profile to "auto-fill" the desired value for an attribute instead of including the attribute in the web form (or when the user leaves the attribute as blank in search). This constraints most often present in hidden databases with social features - dating websites such as match.com and online social networks such LinkedIn fall into this category. With these hidden databases, when a user searches for another one to connect to, the database often use the profile information of the searching user, sometimes without explicitly stating so in the search interface.

Output Interface: A common property shared by the output interfaces of almost almost all hidden databases is a limit on the number of tuples returned (unlike traditional databases where the query returns all matching tuples by default). By the very nature of HTTP-based transmission, the number of tuples loaded at once has to be limited to ensure a prompt response. Of course, a hidden database may support "page down" or "load more" operations that retrieve additional batches of tuples - but the number of such operations allowed for a query is often limited as well (e.g., amazon.com limits the number of page downs to 100). As such, we say that a hidden database enforces a *top-k constraint* when it restricts each query answer to at most k tuples, which can be loaded at once or in multiple batches.

In terms of which (up-to-) k tuples to return, note from the above discussions that we refer to the user-specified conditions as "desired values" rather than "equality predicates". The reason for doing so is because many hidden databases do not limit the returned results to tuples that exactly match all user-specified desired values. Indeed, we can broadly classify hidden database output semantics into two categories, *exact match* and $k\mathrm{NN}$, based on how a user query is answered.

In the exact match scenario, for a given user query q, the hidden database first finds all tuples matching q, denoted as $Sel(q) \subseteq D$. Then, if $|Sel(q)| \leq k$, it simply returns $Sel(q)$ as the query answer. In this case, we say that the query answer is *valid* if $0 < |Sel(q)| \leq k$, or it *underflows* if $Sel(q) = \Omega$. When $|Sel(q)| > k$, however, we say that an *overflow* occurs, and the database has to rely on a proprietary *ranking function* to select k tuples from $Sel(q)$ and return them as the query answer, possibly accompanied by an *overflow flag* - e.g., a warning message such as "your query matched more than k tuples, or an informative message such as "your query matched $|Sel(q)|$ tuples, but only k of them are displayed").

The ranking function used in this exact match scenario may be static or query dependent. Specifically, note that a ranking function can be represented as a scoring function $f(t, q)$ - i.e., for a given query q, the k tuples t with the highest $f(t, q)$ are returned. Here a ranking function is *static* if for a given tuple t, $f(t, q)$ is constant for all queries. An example here is any ranking by a certain attribute - e.g., by price in real-world e-commerce websites. On the other hand, a ranking function is query-dependent if $f(t, q)$ varies for different q - e.g., in flight search, rank by duration if q contains CLASS = BUSINESS, and rank by price if CLASS = ECONOMY.

In the kNN scenario, the hidden database can be considered as simply ranking *all* tuples by a query-dependent ranking function $f(t, q)$ - which essentially (inversely) measures the distance between t and q - and returns the k tuples with the highest $f(t, q)$ (i.e., shortest distances). One can see that, clearly, the returned tuples might not exactly match q on all attributes specified.

Query Rate Limitations: The most critical constraint enforced by hidden databases, impeding our goal of enabling data analytics, is the *query rate* limitation - i.e., a limit on how many search queries a user (or an IP address, an API account, etc.) can issue over a given time period. For example, ebay.com limits the number of queries per IP address per hour to 5000. Since search queries form the only access channel we have over a hidden database, this limit requires us to somehow translate any analytics operations we would like to enable into a small number of search queries, feasible to issue within a short time period even under the query rate limitation.

2.2 Data Analytics over Hidden Databases

In the literature, three common operations over hidden databases have been studied: crawling, sampling and data analytics. The objective of *crawling* is to retrieve all tuples from D. The tuples thus collected can serve as a local repository over which any analytics operation can be performed. The good news about crawling is that its performance boundaries are well understood - [1] reports matching lower and upper bounds (upper to a constant difference) on the number of queries required for crawling a hidden database. The bad news, however, is even the lower bound is too large to be practical under the query rate limitation enforced by real-world hidden databases. As such, crawling is not a focus of the HYDRA system.

Unlike crawling, *sampling* aims to collect only a small subset of tuples $S \subset D$ with $|S| \ll |D|$ - the challenge here is that the subset must be drawn randomly according to a pre-determined sampling distribution. This distribution can be uniform (leading to a simple random sample) or can be weighted based on certain attribute values of interest (leading to a weighted sample). If collected appropriately, the sample is representative of D and can therefore be used for a wide variety of analytics purposes.

Data analytics is the final task which aims to answer aggregate queries (AVG, SUM, COUNT, etc.) over D, so as to enable data mining applications that can be built upon the answered aggregates. As discussed in the introduction, data analytics is better suited when we already know the exact aggregates to be estimated, while sampling is more flexible but may not be as efficient for estimating a given aggregate.

All of these tasks may be carried out over a single snapshot of a hidden database or continuously run when the database changes over time. In the continuous case, the sampling task would need to maintain an up-to-date sample of the current database, while the data analytics task would need to continuously monitor and update the aggregate query answers. In addition, the data analytics task may also need to consider aggregates *across* multiple versions of the hidden database - e.g., the average price cut that happened during the Thanksgiving holiday for all products at a e-commerce website.

Performance Measures: Efficiency-wise, the bottleneck for all the three tasks centers on their *query cost*, i.e., the number of search queries one needs to issue to accomplish a task, because of the aforementioned query rate limitation.

Besides the efficiency measure, sampling and data analytics tasks should also be measured according to the *accuracy* of their outputs. For sampling, the accuracy measure is *sample bias* - i.e. the "distance" between the pre-determined (target) sampling distribution and the actual distribution according to which sample tuples are drawn.

For data analytics, since aggregate query answers, say θ are often estimated by randomized estimators $\tilde{\theta}$, the accuracy is often measured as the expected distance between the two, e.g., the mean squared error (MSE) $MSE(\tilde{\theta}) = E[(\tilde{\theta} - \theta)^2]$, where $E[\cdot]$ denotes the expected value taken over the randomness of $\tilde{\theta}$. Here the MSE actually consists of two components, *bias* and *variance*. Specifically, note that

$$MSE(\tilde{\theta}) = E[(\tilde{\theta} - \theta)^2] = E[(\tilde{(\theta)} - E(\theta))^2] + (E[\tilde{\theta}] - \theta)^2 = Var(\tilde{\theta}) + Bias^2(\tilde{\theta}). \tag{13}$$

An estimator is said to be biased if $E[\tilde{\theta}] \neq \theta$. In contrast, the mean estimate of an unbiased estimator is equal to the true value. The variance of the estimator determines the spread of the estimation (or how the estimate varies from sample to sample). Of course, the goal here is to have unbiased estimators with small variance.

3 Overview of Hydra

Figure 1 depicts the architecture of System HYDRA. It consists of three components: two main components SAMPLE-GEN and SAMPLE-EVAL and one auxiliary component TIMBR. From the perspective of a HYDRA user, the three components work as follows. SAMPLE-GEN enables the user to specify a sampling distribution and then obtain samples according to the specified distribution. SAMPLE-EVAL, on the other hand enables the estimation of a user-specified aggregate query. If the hidden database of concern provides a standardized search query interface, e.g., search APIs such as those provided by Amazon or EBay, then the TIMBR component is not really needed. Otherwise, if only a form-like web interface is provided by the hidden database, the user can use the TIMBR component to easily build a wrapper that translates the search queries required by the two main components to HTTP requests and responses to and from the hidden database server.

Internally, these three components interact with each other (and the remote hidden database) in the following manner. SAMPLE-GEN is responsible for most of the heavy-lifting - i.e., taking a sampling distribution as input and then issuing a small number of search queries to the hidden database in order to produce sample tuples as output. What SAMPLE-EVAL does is to take an aggregate query as input and then select an appropriate sampling distribution based on both the aggregate query and the historic samples it received from SAMPLE-GEN. Once SAMPLE-EVAL receives the sample tuples returned by SAMPLE-GEN, it generates an estimation for the aggregate query answer and then return it to the user. As mentioned above, TIMBR is an optional component

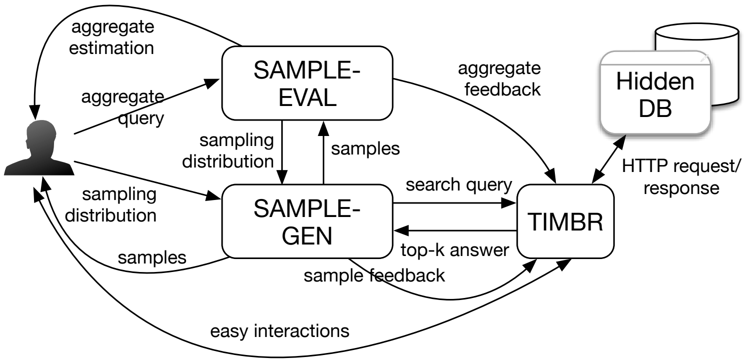


Figure 2: Architecture of HYDRA System

that bridges SAMPLE-GEN and the hidden database server by translating a search query to one or more HTTP requests, and HTTP responses back to the top-k query answers. To do so in an efficient manner, TIMBR solicits easy-to-specify, mouse-click-based, inputs from the user, as well as sample and aggregate feedback from SAMPLE-GEN and SAMPLE-EVAL, respectively. For example, to properly understand the domains of various attributes, it may extract domain values by drawing sample feedback from SAMPLE-GEN, and estimate the popularity of a domain value by drawing aggregate feedback from SAMPLE-EVAL.

4 Component SAMPLE-GEN

In this section, we briefly summarize the key techniques used by SAMPLE-GEN to retrieve samples from underlying hidden web databases. As described in Section 2, we highlight the two most common return semantics used in hidden databases - exact match and kNN. These two models require substantially different techniques. Nevertheless, the estimators used are unbiased, has reasonable query cost and often also has low variance.

4.1 Sample Retrieval for Matching Tuples

We start by describing various techniques to obtain unbiased samples over a hidden databases with a conjunctive query interface and an exact match return semantics. For ease of exposition, we consider a static hidden database with all Boolean attributes and that our objective is to obtain samples with uniform distribution. Please refer to [2, 3] for further generalization and additional optimizations. All the algorithm utilize an abstraction called query tree.

Query Tree: Consider a tree constructed from an arbitrary order of the input attributes $A_1, \dots, A_m$. By convention, we denote the root as Level 1 and it represents the query `SELECT * FROM D`. Each internal (non-leaf) node at level i are labelled with attribute A_i and has exactly 2 outgoing edges (in general U_i edges) one labelled 0 and another labelled 1. Note that each path from the root to a leaf represents a particular assignment of values to attributes. The attribute value assignments are obtained from the edges and each leaf representing potential tuples. In other words, only a subset of the leaf nodes correspond to valid tuples. Figure 3 displays a query tree.

Brute-Force-Sampler: We start by describing the simplest possible sampling algorithm. This algorithm constructs a fully specified query q (i.e. a query containing predicates for all attributes) as follows: for each attribute A_i, it chooses the value v_i to be 0 with probability 0.5 and 1 with probability 0.5. The constructed query is then issued with only two possible outcomes - valid or underflow (since all tuples are distinct). If a tuple is returned, it is picked as a sample. This process is repeated till the requisite number of samples are

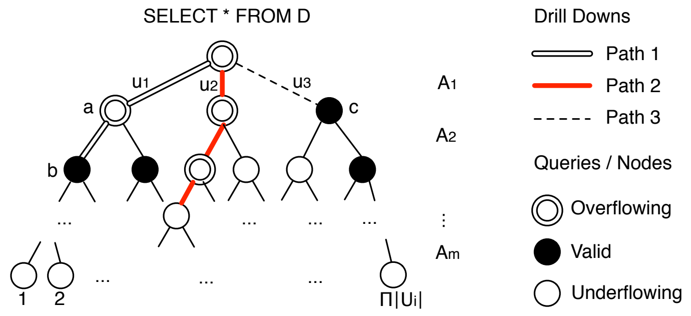


Figure 3: Query Tree

obtained. Notice that this process is akin to picking a leaf node from the query tree uniformly at random. We can see that this process generates unbiased samples. However, this approach is prohibitively expensive as the number of tuples in the database is much smaller than the cartesian product of the attribute values - in other words $n \ll 2^m$. Hence most of the queries will result in underflow necessitating large number of queries to obtain sufficient samples.

Hidden-DB-Sampler: Hidden-DB-Sampler, proposed in [4], takes a slightly more efficient approach. Instead of directly issuing fully specified queries, this sampler performs a fundamental operation known as *random drilldown*. It starts by issuing the query corresponding to the root. At level i, it randomly picks one of the values $v_i \in U_i$ corresponding to attribute A_i and issues the query corresponding to that internal node. Notice that if the query underflows, then the drilldown can be immediately terminated (as further queries will also return empty). If the query is valid, then one of the returned tuples is randomly picked as a sample. If the query overflows, the process is continue till a valid internal or leaf node is reached. We can see that this sampler requires less query cost than Brute-Force-Sampler. However, the efficiency comes at the cost of skew where the tuples no longer are reached with same probability. Specifically, note that "short" valid queries are more likely to be reached than longer queries. A number of techniques such as acceptance/rejection sampling could be used to fix this issue. The key idea is to accept a sample tuple with probability proportional to $1/2^l$ where l is the length/number of predicates in the query issued. However, it is possible that a number of tuples could be rejected before a sample is accepted.

Samplers that Leverage Count Information: As pointed out in Section 2, most real-world hidden web databases provide some feedback about whether a query overflows by either alerting the user or providing the number of matching tuples for each query. It is possible to leverage the count information to design more efficient samplers.

Let us first consider the scenario where the count information is available. Count-Decision-Tree-Sampler, proposed in [5], uses two simple ideas to improve the efficiency of Hidden-DB-Sampler. First, it leverages *query history* - the log of all queries issued, their results and the counts. This information is then used in preparing which query to execute next. Specifically, given a set of candidate queries to choose from, the sampler prefers the query that already appears in the history which results in substantial query savings. Another idea is to generalize the notion of attribute ordering used in query tree. While the previous approaches used an arbitrary but fixed ordering, additional efficiency can be obtained by incrementally building the tree based on a saving function that helps us choose queries that have a higher chance of reaching a random tuple soon. The algorithm for retrieving unbiased samples where only an alert is available is bit trickier. Alert-Hybrid-Sampler operates in two stages. First, it obtains a set of *pilot* samples that is used to estimate the count information for queries. In the second stage, we reuse the algorithm that leverages count information. While this results in an inherent approximation and bias, it is often substantially more efficient in terms of number of queries.

HD-Unbiased-Sampler: This sampler, proposed in [2], is one of the state of the art samplers that uses a number of novel optimizations for better efficiency. We discuss the fundamental idea here and the various variance reduction techniques in Section 5. In contrast to other techniques that obtain uniform random samples with unknown bias, this sampler intentionally seeks biased samples - albeit those for which the bias is precisely known. This additional knowledge allows us to "correct" the bias and obtain unbiased samples.

The random drilldown process is very similar to the previous samplers. We start with the root node and choose an edge uniformly at random. If the query overflows, the process is continued. If it is valid, we can terminate the process by choosing a tuple uniformly at random from the results. When the query underflows, instead of restarting the drill down, this sampler instead backtracks. Specifically, it backtracks to a sibling node (i.e. a node with which it shares the parent node) and continue the drill down process. We can note that the drill down always terminates with a node whose parent is overflowing. By using few additional queries, [2] proposed an efficient estimator that can precisely compute the probability that an overflowing node is visited. If the ultimate purpose is to estimate aggregates, one can use Horvitz-Thompson estimator to generate estimates based on the known bias of each tuple.

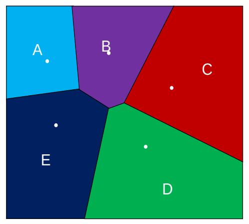
4.2 Sample Retrieval for kNN Model

In this subsection, we examine the kNN return semantics where we are given an arbitrary query q and the system returns k tuples that are nearest to q based on a scoring (or distance) function. The objective is to design efficient techniques for obtaining uniform random samples given a kNN query interface. We focus our attention to a class of hidden databases (spatial databases) and describe recent attempts to retrieve uniform samples. Specifically, we focus on *location based services* (LBS) that have become popular in the last few years. Websites such as Google Maps and other social networks such as WeChat extensively use this return semantics. Given a query point (location) q, an LBS returns k points from D that are closest to q. Note that the hidden database consists of 2-dimension points represented as latitude/longitude which can correspond to points of interest (such as in Google Maps) or user locations (such as in WeChat, Sina Weibo etc). For this article, we assume that the distance function used in Euclidean distance and that the objective is to obtain uniform samples.

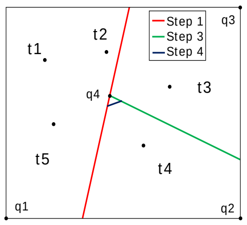
Taxonomy of LBS: Based on the amount of information provided for each query, each LBS can be categorized into two classes. A Location-Returned-LBS (LR-LBS) returns the precise location for each of the top-k returned tuples[6]. A number of examples such as Google Maps, Bing Maps, Yelp follow this model as they display the location information of the POIs such as restaurants. A Location-Not-Returned-LBS (LNR-LBS), on the other hand, does not return tuple locations[6]. This is quite common among social networks with some location based functionality such as Weibo due to privacy concerns. Both these categories require substantially different approaches for generating samples with divergent efficiencies. As we shall see later, the availability of location dramatically simplifies the sampling design.

Voronoi Cells: Assume that all points in the LBS are contained in a bounding box B. For example, all POIs in USA can be considered as located in a bounding box encompassing USA. The Voronoi cell of a tuple t, denoted by $V(t)$ is the set of points on the B-bounded plane that are closer to t than any other tuple in D[7]. In other words, given a Voronoi cell of a tuple t, all points within it return t as the nearest neighbor. We can notice that the Voronoi cells are mutually exclusive and also convex[7]. From the definition, we can notice that the ratio of area of the Voronoi cell to the area of the bounding box B precisely provides the probability that a given tuple is the top ranked result. Figure 4a displays the Voronoi diagram for a small database.

Overview of Approach: At a high level, generating samples for LR-LBS and LNR-LBS follow a similar procedure. In both cases, we generate random point queries, issue them over the LBS and retrieve the top ranked tuple. Note that based on the distribution of underlying tuples, some tuples may be retrieved at a higher rate. For example, a POI in a rural area might be returned as a top result for a larger number of query points than a POI in a densely populated area. Hence, the process of randomly generating points and issuing them results in a biased distribution for the top ranked tuples. However, using the idea from HD-Unbiased-Sampler, we can correct the bias if we know the probability that a particular POI will be returned as the top result. For this purpose, we



(a)



(b)

Figure 4: Voronoi Diagram and Illustration of LR-LBS-AGG

use the concept of Voronoi cells[7] that provides a way to compute the bias. [6] proposes methods for precise computation of Voronoi cell of a tuple for LR-LBS. For LNR-LBS, [6] describes techniques to compute Voronoi cell of a tuple to arbitrary precision.

Algorithms LR-LBS-AGG and LNR-LBS-AGG: This algorithm generates samples and uses them for aggregate estimation over LR-LBS such as Google Maps. Note that in this case, the location of the tuple is also returned. The key idea behind computation of the Voronoi cell of a tuple is as follows: if we somehow collect all the tuples with Voronoi cell adjacent to t, then we can precisely compute Voronoi cell of t[7]. So the challenge now reduces to designing a mechanism to retrieve all such tuples. A simple algorithm (see [6] for further optimizations) is as follows. We start with a singleton set D' containing t with its potential Voronoi cell containing the whole bounding box B. We issue queries corresponding to the boundaries of B. If any query returns an unseen tuple, we append it to D', recompute the Voronoi cell and repeat the process. If all the boundary points return a tuple that we have already seen, then we can terminate the process. This simple procedure provides an efficient method to compute the Voronoi cell and thereby the bias of the corresponding tuple. We again use Horvitz-Thompson estimator for estimating the aggregate by assigning unequal weights to tuples based on their area. The key challenge for applying this approach to LNR-LBS is that the location of a tuple is not returned. However, [6] proposes a "binary search" primitive for computing the Voronoi cell of a tuple t. When invoked, this primitive identifies one edge of the Voronoi cell of t to arbitrary precision as required.

Figure 4b provides a simple run-through of the algorithm for a dataset with 5 tuples $\{t_1, \ldots, t_5\}$. Suppose we wish to compute $V(t_4)$. Initially, we set $D' = \{t_4\}$ and $V(t_4) = V_0$, the entire region. We issue query q_1 that returns tuple t_5 and hence $D' = \{t_4, t_5\}$. We now obtain a new Voronoi edge that is the perpendicular bisector between t_4 and t_5. The Voronoi edge after step 1 is highlighted in red. In step 2, we issue query q_2 that returns t_4 resulting in no update. In step 3, we issue query q_3 that returns t_3. $D' = \{t_3, t_4, t_5\}$ and we obtain a new Voronoi edge as the perpendicular bisector between t_3 and t_4 depicted in green. In step 4, we issue query q_4 that returns t_2 resulting in the final Voronoi edge depicted in blue. Further queries over the vertices for $V(t_4)$ does not result in new tuples concluding the invocation of the algorithm.

5 Component SAMPLE-EVAL

In this section, we describe the techniques used in the SAMPLE-EVAL component that is used to evaluate the quality of samples collected by SAMPLE-GEN component. In this article, we focus our evaluation of samples based on the end-goal of aggregate estimation. We consider three major factors: bias, variance and longevity. We start by describing bias and variance in in Section 5.1. Since there exist unbiased estimators for common aggregates, we focus on known techniques to reduce the variance of an estimator. In Section 5.2, we describe a principled method for updating sample when the underlying database changes.

5.1 Static Database: Bias and Variance

Recall from Section 2 that the *bias* of an estimator is the difference between the expected value and the true value of the aggregate. In other words, it provides information about whether the expected value of an aggregate (estimated from the samples) agrees with the actual value. It is preferable for an estimator to be unbiased - i.e. have a bias of 0. The *variance* of an estimator measures how adjusting the sampling distribution affects the variance of estimated aggregates - in other words, how aggregate estimated varies from sample to sample.

If an estimator has no bias, then the long term average of the estimates converges to the true value. If the estimator has a small but known bias, then the estimates converges around the biased value. If the estimator has a low variance, most estimates generated from samples cluster around the expected value of the estimator. Finally, if the estimator has a high variance, the estimates generated can vary widely around the expected value. It is possible to have an estimator with large bias and low variance and vice versa.

Recall from Section 2 that bias and variance are tightly interconnected through the MSE of an estimator. If all things being equal, an unbiased estimator is preferable to a biased estimator. If the estimator is unbiased, then it is preferable to have a low variance. Finally, it might not always be possible to design an unbiased estimator - there are a number of practical estimators that are biased but with a small bias. The relevant trade-off between bias and variance is often specific to the end goal.

There exist a number of estimators that provide unbiased estimation of aggregates over hidden databases [2, 3, 4]. Hence in this section, we focus on some principled mechanisms to reduce the variance. For additional details, please refer to [8].

Acceptance/Rejection Sampling: Recall from Section 4 that Hidden-DB-Sampler sought to improve Brute-Force-Sampler by issuing shorter (in terms of number of predicates) and broader (as measured by the subset of database matched) queries. Basically, the sampler performs a random drill down starting from the root. When it reaches an underflowing node, it restarts the drill down. For overflowing nodes, it continues the drill down process while for a valid node, it takes one of the returned tuples as a sample. We can see that this process introduces a bias towards tuples that are favored by the ranking function (and hence returned by shorter queries). A common technique used to correct this is called acceptance rejection sampling[9]. Instead of unconditionally accepting a tuple from a valid query, we discard the tuples that were obtained from a short random drill down with a higher probability. If done correctly, the rejection results in a set of accepted samples that approximate the uniform distribution.

Weighted Sampling: While acceptance/rejection sampling provides a uniform random samples, they often reject a disproportionate number of samples. A major improvement can be achieved by *weighted sampling*[9] that *accepts* all the tuples from the drill down and instead associates with each of them a weight that is proportional to its selection probability $p(q)$ (probability that the tuple is selected). Now, we can generate a more robust estimate by adjusting each estimate by its selection probability. This is often achieved through the unequal probability sampling estimators such as Horvitz-Thompson[9]. In addition to resulting in substantial savings of query cost (as no sample is rejected), the estimator remains unbiased. Of course, it is now necessary to design additional mechanisms to estimate the selection probability as accurate as possible as it has a disproportionate impact over the estimator accuracy. This approach often works well if the accuracy of answering an aggregate query depends on whether the distribution of selection probability of tuples is aligned with the aggregate to be estimated. Please refer to [10] for additional discussions.

Weight Adjustment: This a popular variance reduction technique that has broad applicability in both traditional and hidden database sampling. The key idea here is to avoid the potential misalignment between aggregate and selection probability distribution by proactively seeking to "align" these factors. Specifically, this is achieved by adjusting the probability for following each branch in a query tree. Such a change affects the selection probability of nodes and potentially results in a better alignment. Achieving perfect alignment is often hard and would make the estimator too specific to an aggregate. Instead, it is possible to design an effective heuristics that has generality and often approximates the perfect alignment with reasonable fidelity. The typical strategy to perform weight alignment is to obtain a set of "pilot" samples and use them to estimate the measure attribute distribution. From the pilot samples, we can estimate for each branch the selection probability proportional to the size of the database subset rooted under this branch. While knowledge of the exact sizes results in 0 variance, the approximation via pilot samples often reduces the variance in practice. Please refer to [2] for additional discussions.

Crawling of Low-Probability Zones: There exists a number of well known samplers [5] that could leverage overflow information to improve the efficiency of sampling. This is achieved by having a constant cutoff C that trades-off the balance between sampling efficiency and variance. These samplers assume that all queries at Level-C results in underflow or is valid and choose branch with different probabilities. Such an approach has the unfortunate side-effect of rarely choosing tuples that are only returned by leaf level query nodes. This skew between tuples that are reachable at low and high levels can be mitigated by combining sampling with crawling of low probability zones. Specifically, the sampling of tuples below Level-C starts when the sampler reaches an overflow query Q at level C but could not select a sample from it (possibly due to rejection sampling)[10]. Instead of continuing the drill down below level-C, we can crawl the entire sub-tree rooted at Q and use the results to estimate the size of tuples in the sub-tree that match Q. A tuple randomly chosen from the set of tuples that match Q is returned as sample. We can see that this modification reduces the skew of the selection probability of tuples below Level-C.

5.2 Dynamic Databases: Longevity

In this subsection, we briefly discuss the third factor in reducing estimation variance by considering "longevity". The approaches discussed in previous section are applicable only for static databases. However, most real world databases often undergoes updates (inserts, modifications and deletions). Longevity is a key factor in the determining how a sample tuple retrieved using one of the prior rounds must be handled.

A straightforward, but naive, approach for handling dynamic databases is "repeated execution" (RESTART-ESTIMATOR) where samplers for static databases are invoked after each round. However, such an approach has number of issues. First, the query budget for each round is spent wholly on retrieving new samples for that round. This causes the variance of the aggregate estimation in each round to remain more or less constant (as it in turn inversely related to the number of samples). We can readily see that it must be possible to have a better process that carefully apportions the query budget between updating samples from prior rounds and retrieving new samples.

A straightforward improvement is REISSUE-ESTIMATOR[3] that outperforms the naive algorithm by leveraging historic query answers. Specifically, any random drill down conducted is uniquely represented by a "signature". After the first round, the signature and results of the drill down are stored. In the subsequent rounds, we reuse the same set of signatures (as against the naive algorithm that generates new drill downs). Recall that all the samplers for static databases terminated the drill down when they reached a valid node. Hence, in each subsequent round, we can start our process from the signature node. If the node overflows (new tuples added), then we drill down. If it underflows (tuples deleted), then drill up. This process is repeated until a valid node is reached when the sample is replaced by a randomly chosen tuple from its results. This estimator is still unbiased and can utilize the queries saved by reusing the signatures for obtaining additional samples in each round. Note that when database does not change between successive rounds, the queries issued by REISSUE-ESTIMATOR are always a subset of those issued by RESTART-ESTIMATOR resulting in substantial query savings. In the worst case, when the database is completely recreated in each round, REISSUE-ESTIMATOR *might* end up performing worse than RESTART-ESTIMATOR. In practice, REISSUE-ESTIMATE often results in query savings that could be used to retrieve new samples in that round. The new samples results in reduction of variance (as variance of the aggregate is inversely proportional to number of samples).

This idea could be extended further to design an even more sophisticated RS-ESTIMATOR[3] that distributes the query budget available for each round into two parts: one for reissuing (i.e., updating) drill downs from previous rounds, and the other for initiating new drill downs. Intuitively, the distribution must be computed based on the amount of changes in the aggregate to be estimated. [3] describes the relevant optimization information that could be used for the distribution of query cost that are usually approximated via pilot samples. Intuitively, when the database undergoes little change, RS-ESTIMATOR mostly conducts new drill downs - leading to a lower estimation error than REISSUE and when the database changes drastically, RS-ESTIMATOR will be automatically reduced to REISSUE-ESTIMATOR[3].

6 Component TIMBR

In this section, we describe the TIMBR component of HYDRA which is responsible for three tasks: (1) translating a search query supported by a hidden database to the corresponding HTTP request that can be submitted to the website; (2) interpreting a webpage displaying the returned query answer to extract the corresponding tuple values; and (3) the proper scheduling of queries (i.e., HTTP requests submitted to websites), especially when there is a large number of concurrent data analytics tasks running in the HYDRA system. The following three subsections depict our design of the three components, respectively. As we mentioned in the introduction, it is important to understand that our goal here is not to develop new research results for the vast field of information extraction, but to devise a simple solution that fits our goal of the HYDRA system - i.e., a tool that a data analyst with substantial knowledge of the underlying data (as otherwise he/she would not be able to specify the aggregate queries anyway) can quickly deploy over a form-like hidden database. One can see in the following discussions how our design of TIMBR leverages two special properties of our system - the auxiliary input from a human data analyst; and the (bootstrapped) feedback from the SAMPLE-GEN and SAMPLE-EVAL components.

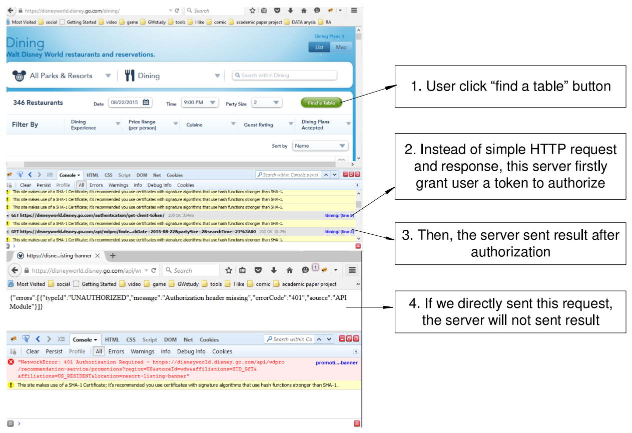
6.1 Input Interface Modeling

We start with the first task of TIMBR, input interface modeling, i.e., how to translate a search query supported by a hidden database to a corresponding HTTP request that can be transmitted to the web server of the hidden database. Before describing our design, we first outline two main technical challenges facing the modeling of an input interface: First is the increasingly complex nature of the (output) HTTP request. Traditionally (e.g., era before 2008), form-like search interfaces were usually implemented as simple HTML forms, with input values (i.e., search predicates) transmitted as HTTP GET or POST parameters. For these interfaces, the only thing our input-interface modeling component needs to identify is a URL plus a specification of how each attribute (i.e., predicate) is mapped to a GET or POST parameter. Here is an example:

URL: `http://www.autodb.com/`
Parameter: Make = p1 (GET), Model = p2 (GET)
Composed HTTP GET request: `http://www.autodb.com/search?p1=ford&p2=F150`

Nonetheless, more recently designed websites often use complex Ajax (i.e., asynchronous JavaScript) requests to transmit search queries to the web server - leading to sometimes not one, but multiple HTTP requests and responses that are interconnected with complex logic. For example, Figure 5 demonstrates an example with disney.com, which uses one Ajax request to retrieve an authentication token before using it in a second Ajax request to obtain the real query answer.

One can see that, for websites with such complex designs, it becomes impossible to properly model its input interface without parsing through the corresponding JavaScript source code. The second challenge stems from the first one, and is also an implication of the more powerful and flexible HTML5/JavaScript designs that are rapidly gaining popularity. Note that if the first challenge did not exist - i.e., if the output of interface modeling were simply an HTTP GET or POST request as in the above example - then we would not care about the design of the input (i.e., search interface) webpage at all. The reason is that, whatever the design might be, we only need to capture the submission URL and figure out the mapping between attributes and



1. User click "find a table" button

2. Instead of simple HTTP request and response, this server firstly grant user a token to authorize

3. Then, the server sent result after authorization

4. If we directly sent this request, the server will not sent result

```
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
GET https://disneyworld.disney.go.com/authentication/get-client-token/  200 OK 334ms    /dining/ (line 8)
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
GET https://disneyworld.disney.go.com/api/wdpro/finde...chDate=2015-08-22&partySize=2&searchTime=21%3A00  200 OK 15.29s    /dining/ (line 8)
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
```

```
{"errors":[{"typeId":"UNAUTHORIZED","message":"Authorization header missing","errorCode":"401","source":"API Module"}]}
```

```
"NetworkError: 401 Authorization Required - https://disneyworld.disney.go.com/api/wdpro
/recommendation-service/promotions?region=US&storeId=wdw&affiliations=STD_GST&
affiliations=US_RESIDENT&location=resort-listing-banner"    promoti...-banner
This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1.
```

Figure 5: Example of multiple HTTP requests/responses

GET/POST parameters. However, given the presence of the first challenge and the necessity of "understanding" the JavaScript code required for query submission, it becomes necessary for us to identify the interface elements (and correspondingly, the JavaScript variables) capturing the user-specified values for each attribute.

It is from this requirement that the second challenge arises. Here the difficulty stems from the complex design of input controls e.g., textboxes or dropdown menus for specifying search predicates. Take textbox as an example. Traditionally, it is often implemented as a simple <input> tag that can be easily identified and mapped to an attribute according to minimal user interactions. Now, however, many websites such as Walmart implement textboxes in a completely customized manner, e.g., as a simple <div> that responds to mouse clicks and/or keyboard events using custom-designed functions. Once again, this makes JavaScript parsing a prerequisite for input interface modeling.

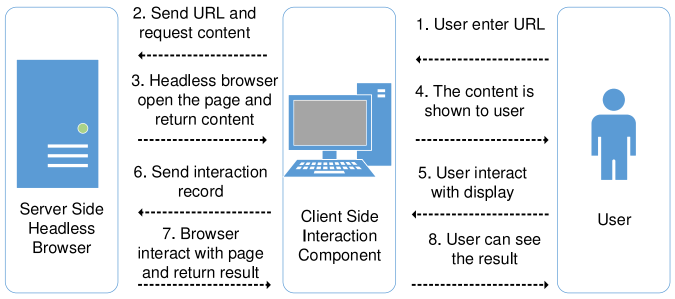

Server Side Headless Browser
Client Side Interaction Component
User
1. User enter URL
2. Send URL and request content
3. Headless browser open the page and return content
4. The content is shown to user
5. User interact with display
6. Send interaction record
7. Browser interact with page and return result
8. User can see the result

Figure 6: Process of Timbr

In light of the two challenges, our main technique for input interface modeling is not to parse the HTML and JavaScript code in our system, but to instead leverage an existing headless browser, e.g., PhantomJS, which runs on our HYDRA server. Figure 6 demonstrates the process of constructing an input-interface model. We start by doing two things: starting a headless browser at the backend and displaying the target website (i.e., the hidden database) as part of the HYDRA interface at the frontend. Then, we ask the user to interact with the displayed website to specify the input search conditions. The exact user interactions with the display (i.e., mouse clicks, keyboard events, etc.) are captured and transmitted to our backend headless browser, which repeats these interactions and (in the future) adjusts them - e.g., by changing the value specified for an attribute to form other search queries.

One can see here a key premise of our design: to convert the problem of modeling "how the input webpage interacts with the hidden database server" - a tedious task that requires complex HTML/JavaScript parsing -

to simply model "how the user interacts with the input webpage" - a much simpler task given its (general) transparency to the website Ajax design, yet can solve the same problem with the help of a headless browser running on HYDRA server. Of course, our approach calls for more user inputs than what is required by the fully automated interface modelers proposed in the literature (like [11, 12]) - e.g., instead of using techniques such as image pattern recognition [13] to automatically identify attribute controls, the mapping between an attribute and its corresponding webpage control (i.e., interaction) is defined by the human user in our TIMBR component. Nonetheless, we note that, as discussed in the introduction, our solution here is simpler and fits our goal of developing a fast-prototyping module for data analysts who already have substantial knowledge of the domain of the underlying data.

The detailed implementation calls for the proper treatment of many subtle issues. Some examples include how to properly measure the delay required between two operations (e.g., when the first operation activates an input control and the second one specifies the input value), how to detect and identify the correlation between different operations (e.g., only when car make is selected will a drop-down menu of car model appear), etc. Due to the space limitation here, we refer readers to our system documentation for details of these design issues [14].

6.2 Output Modeling

Compared with the input side, output modeling is a simpler task as the goal here is mainly to extract information from the returned result page, with only a few user interactions to capture. In the following discussions, we first describe our techniques for extracting tuples and attributes from the webpage, and then briefly discuss the interactions that need to be captured.

Extracting Tuples and Attributes: Consider the Document Object Model (DOM) tree of the result webpage. The goal here is to identify the group of subtrees corresponding to the tuples being returned and (at a finer granularity) the position of each attribute within a subtree. Ideally, all tuples should be mapped to the same level on the DOM tree, making the identification a straightforward process - so long as the user specify (by clicking on the HTML element corresponding to) the subtree of one example tuple, the remaining job is to simply find all other subtrees at the same level.

The practical cases are often more complex. For example, some websites place a subset of tuples (often those "featured" ones) into a special element, making them one level lower than the rest of tuples. In TIMBR, we address these challenges with two main ideas, namely human- and data-assisted tuple specification, respectively: the human-assisted approach asks a user to specify two tuples (ideally one at the beginning and the other at the end), in order to find the correct common ancestor HTML element for all tuples. Then, based on auxiliary information such as common CSS classes, the identification of tuple subtrees under this common ancestor becomes straightforward. The data-assisted approach is mainly used when such auxiliary information is not available. In this case, we mark as the "signature" of each subtree the positions and types of its elements that are identifiable using pre-known attribute domains (e.g., ZIP codes in US are five-digit combinations, US phone numbers are of the format 3 digits + 3 digits + 4 digits). Such a signature is then used to properly identify the subtrees under the common ancestor that are corresponding to the returned tuples.

Interaction Modeling: The interactions we need capture from the output interface are those "next page" or "load more" operations - they retrieve additional tuples that cannot be returned on the current page. Given the popularity of "infinite scroll" - i.e., Ajax-based loading of additional tuples without refreshing URL - in recently designed websites, we once again use the headless browser described in the input modeling part to capture the interaction and trigger it to retrieve additional tuples. A challenge with this design, however, is duplicate prevention. Specifically, some websites (e.g., *http://m.bloomingdales.com*) make the "next page" button available even when the returned results have already been exhausted. When a user clicks on the next page button, the same tuples will be displayed, leading to duplicate results. Note that we cannot simply compute a hash of the entire webpage HTML for duplicate detection, as peripheral contents (e.g., advertisements) on the page might change with each load. Thus, in TIMBR we first extract tuple and attribute values from each page, and then compare the extracted values against the history to identify any duplications.

7 Related Work

Traditional Database Sampling and Approximate Query Processing: Sampling in traditional databases have a number of applications such as selectivity estimation, query optimization, approximate query processing etc. [15] proposed some of the earliest methods for enabling sampling in a database that has been extended by subsequent work. Most uniform sampling approaches suffer from low selectivity problem which is often addressed by biased samples. A number of weighted sampling techniques [16, 17] has been proposed. In particular, [17] exploits workload information to continuously tune the samples. Additional techniques for overcoming limitations of sampling such as outlier indexing[18], stratified sampling[19], dynamic sampling selection[20] has also been proposed. [21] proposed an online sampling technique with a time-accuracy tradeoff while [22] extended it by introducing novel methods of joining database tables.

There has been extensive work on approximate aggregate query processing over databases using sampling based techniques [23, 24, 25] and non sampling based techniques such as histograms [26] and wavelets [27]. Please refer to [28] for a survey. A common technique is to build a synopsis data structure that is a concise yet reasonably accurate representation of the database which can then be used to perform aggregate estimation. Maintenance of statistical aggregates in the presence of database updates have been considered in [29, 30, 31].

Hidden Database Crawling, Sampling and Analytics: The problem of crawling a hidden structured database has been extensively studied [32, 33, 34, 35, 36, 1]. The early works focussed on identifying and formulating effective queries over HTML forms so as to retrieve as many tuples as possible. In particular, [35] proposed an highly automatic and scalable mechanism for crawling hidden web databases by choosing an effective subset of search space of all possible input combinations that returns relevant tuples. [1] proposed optimal algorithms to completely crawl a hidden database with minimum queries and also provided theoretical results for the upper and lower bounds of query cost required for crawling.

There has been a number of prior work in performing sampling and aggregate estimation over hidden databases. [4, 5, 10] describe efficient techniques to obtain random samples from hidden web databases that can then be utilized to perform aggregate estimation. [2] provided an unbiased estimator for COUNT and SUM aggregates for databases with form based interfaces. [37] proposed an adaptive sampling algorithm for aggregation query processing over databases with hierarchical structure. Efficient algorithms for aggregate estimation over dynamic databases was proposed in [3]. There has been a number of prior work on enabling analytics over structured hidden databases such as generating content summaries[38, 39, 40], processing top-k queries[41], frequent itemset mining[42, 43], differential rule mining[44]. A number of techniques for variance reduction for analytics such as weight adjustment[2], divide-and-conquer[2], stratified sampling[42] and adaptive sampling[43, 45] has also been proposed.

Search Engine Sampling and Analytics: A number of prior work have considered crawling a search engine's corpus such as [46, 47, 48, 49] where the key objective was discovering legitimate query keywords. Most existing techniques over a search engine's corpus require prior knowledge of a query pool . Among them, [50] presented a method to measure the relative sizes between two search engines' corpora. [51, 52] achieved significant improvement on quality of samples and aggregate estimation. [53] introduced the concept of designated query, that allows one to completely remove sampling bias. Sampling databases through online suggestions was discussed in [54]. The key issue with requiring a query pool is that constructing a "good" query pool requires detailed knowledge of the database such as its size, topic, popular terms etc. There exist a handful of papers that mine a search engine's corpus without a query pool [51, 55] that operate by constructing a document graph (not fully materialized) and performing an on-the-fly random walks over them.

Information Extraction: There has been extensive prior work on information integration and extraction over hidden databases - see related tutorials [56, 57]. Parsing and understanding web query interfaces has been extensively studied (e.g., [13, 58]). The mapping of attributes across different web interfaces has been studied (e.g., [59]) while techniques for integrating query interfaces for multiple web databases can be found in [60, 61].

8 Final Remarks and Future Work

In this paper, we provided an overview of System HYDRA which enables fast sampling and data analytics over a hidden web database that provides nothing but a form-like web search interface as its only access channel. Specifically, we discussed three key components of HYDRA: SAMPLE-GEN which produces samples according to a given sampling distribution, SAMPLE-EVAL which generates estimations for a given aggregate query, and TIMBR which enables the fast and easy construction of a wrapper that translates a supported search query to HTTP requests and retrieves top-k query answers from HTTP responses.

It is our belief and vision that HYDRA enables a wide range of future research on hidden web databases. Within the database community, we think the research of HYDRA raises a fundamental question: exactly what kind of information can be inferred from a search-query-only, top-k-limited, access interface? HYDRA provides part of the answer by demonstrating that samples and aggregate query answers can be inferred from such an interface. But more research is needed to determine what other information can be inferred - and for other types of access interfaces/channels as well. Outside of the database community, we believe HYDRA provides a platform that enables inter-disciplinary studies with domain experts from other fields such as economy, sociology, etc., to investigate and understand the vast amount of data in real-world hidden web databases. It is our hope that HYDRA marks the start of an era that witnesses the effective and wide-spread usage of valuable analytics information in the deep web.

References

[1] C. Sheng, N. Zhang, Y. Tao, and X. Jin, "Optimal algorithms for crawling a hidden database in the web," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1112–1123, 2012.

[2] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*, 2010.

[3] W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimation over dynamic hidden web databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1107–1118, 2014.

[4] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[5] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009.

[6] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimations over location based services," *PVLDB*, vol. 8, no. 12, pp. 1334–1345, 2015.

[7] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.

[8] N. Zhang and G. Das, "Exploration of deep web repositories," in *Tutorial, VLDB*, 2011.

[9] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[10] A. Dasgupta, N. Zhang, and G. Das, "Turbo-charging hidden database samplers with overflowing queries and skew reduction," in *EDBT*, 2010.

[11] V. Crescenzi, G. Mecca, P. Merialdo *et al.*, "Roadrunner: Towards automatic data extraction from large web sites," in *VLDB*, vol. 1, 2001, pp. 109–118.

[12] G. Gkotsis, K. Stepanyan, A. I. Cristea, and M. Joy, "Entropy-based automated wrapper generation for weblog data extraction," *World Wide Web*, vol. 17, no. 4, pp. 827–846, 2014.

[13] E. Dragut, T. Kabisch, C. Yu, and U. Leser, "A hierarchical approach to model web query interfaces for web source integration," in *VLDB*, 2009.

[14] "TIMBR system documentation," `http://timbr.me/documents/guide.pdf`.

[15] F. Olken, "Random sampling from databases," Ph.D. dissertation, University of California at Berkeley, 1993.

[16] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 487–498.

[17] V. Ganti, M.-L. Lee, and R. Ramakrishnan, "Icicles: Self-tuning samples for approximate query answering." in *VLDB*, vol. 176. Citeseer, 2000, p. 187.

[18] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, "Overcoming limitations of sampling for aggregation queries," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 534–542.

[19] S. Chaudhuri, G. Das, and V. Narasayya, "A robust, optimization-based approach for approximate answering of aggregate queries," in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 295–306.

[20] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 539–550.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 171–182, 1997.

[22] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *ACM SIGMOD Record*, vol. 28, no. 2. ACM, 1999, pp. 287–298.

[23] S. Chaudhuri, G. Das, and V. R. Narasayya, "A robust, optimization-based approach for approximate answering of aggregate queries." in *SIGMOD*, 2001.

[24] S. Chaudhuri, G. Das, and V. R. Narasayya, "Optimized stratified sampling for approximate query processing." *ACM Trans. Database Syst.*, vol. 32, no. 2, p. 9, 2007.

[25] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya, "Overcoming limitations of sampling for aggregation queries," in *ICDE*, 2001.

[26] V. Poosala and V. Ganti, "Fast approximate query answering using precomputed statistics." in *ICDE*, 1999, p. 252.

[27] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," in *The VLDB Journal*, 2000, pp. 111–122.

[28] M. N. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes," in *VLDB*, 2001.

[29] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets," in *SODA*, 1999, pp. 909–910.

[30] J. Gehrke, F. Korn, and D. Srivastava, "On computing correlated aggregates over continual data streams," in *SIGMOD*, 2001, pp. 13–24.

[31] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *SIGMOD*, 2002.

[32] M. Álvarez, J. Raposo, A. Pan, F. Cacheda, F. Bellas, and V. Carneiro, "Crawling the content hidden behind web forms," *Computational Science and Its Applications–ICCSA 2007*, pp. 322–333, 2007.

[33] A. Calì and D. Martinenghi, "Querying the deep web," in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 724–727.

[34] S. W. Liddle, D. W. Embley, D. T. Scott, and S. H. Yau, "Extracting data behind web forms," in *Advanced Conceptual Modeling Techniques*. Springer, 2003, pp. 402–413.

[35] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's deep web crawl," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1241–1252, 2008.

[36] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 129–138.

[37] F. N. Afrati, P. V. Lekeas, and C. Li, “Adaptive-sampling algorithms for answering aggregation queries on web sites,” *DKE*, vol. 64, no. 2, pp. 462–490, 2008.

[38] J. Callan and M. Connell, “Query-based sampling of text databases,” *ACM TOIS*, vol. 19, no. 2, pp. 97–130, 2001.

[39] P. Ipeirotis and L. Gravano, “Distributed search over the hidden web: Hierarchical database sampling and selection,” in *VLDB*, 2002.

[40] Y.-L. Hedley, M. Younas, A. E. James, and M. Sanderson, “Sampling, information extraction and summarisation of hidden web databases,” *DKE*, vol. 59, no. 2, pp. 213–230, 2006.

[41] N. Bruno, L. Gravano, and A. Marian, “Evaluating top-k queries over web-accessible databases,” in *International Conference on Data Engineering*, 2002.

[42] T. Liu, F. Wang, and G. Agrawal, “Stratified sampling for data mining on the deep web,” *Frontiers of Computer Science*, vol. 6, no. 2, pp. 179–196, 2012.

[43] F. Wang and G. Agrawal, “Effective and efficient sampling methods for deep web aggregation queries,” in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 2011, pp. 425–436.

[44] T. Liu and G. Agrawal, “Active learning based frequent itemset mining over the deep web,” in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 219–230.

[45] T. Liu, F. Wang, and G. Agrawal, “Stratified sampling for data mining on the deep web,” *Frontiers of Computer Science*, vol. 6, no. 2, pp. 179–196, 2012.

[46] E. Agichtein, P. G. Ipeirotis, and L. Gravano, “Modeling query-based access to text databases,” in *WebDB*, 2003.

[47] A. Ntoulas, P. Zerfos, and J. Cho, “Downloading textual hidden web content through keyword queries,” in *JCDL*, 2005.

[48] L. Barbosa and J. Freire, “Siphoning hidden-web data through keyword-based interfaces.” in *SBBD*, 2004, pp. 309–321.

[49] K. Vieira, L. Barbosa, J. Freire, and A. Silva, “Siphon++: a hidden-webcrawler for keyword-based interfaces,” in *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 2008, pp. 1361–1362.

[50] K. Bharat and A. Broder, “A technique for measuring the relative size and overlap of public web search engines,” in *WWW*, 1998.

[51] Z. Bar-Yossef and M. Gurevich, “Random sampling from a search engine’s corpus,” *Journal of the ACM*, vol. 55, no. 5, 2008.

[52] Z. Bar-Yossef and M. Gurevich, “Efficient search engine measurements,” in *WWW*, 2007.

[53] M. Zhang, N. Zhang, and G. Das, “Mining a search engine’s corpus: efficient yet unbiased sampling and aggregate estimation,” in *SIGMOD*, 2011, pp. 793–804.

[54] Z. Bar-Yossef and M. Gurevich, “Mining search engine query logs via suggestion sampling,” in *VLDB*, 2008.

[55] M. Zhang, N. Zhang, and G. Das, “Mining a search engine’s corpus without a query pool,” in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 29–38.

[56] K. Chang and J. Cho, “Accessing the web: From search to integration,” in *Tutorial, SIGMOD*, 2006.

[57] A. Doan, R. Ramakrishnan, and S. Vaithyanathan, “Managing information extraction,” in *Tutorial, SIGMOD*, 2006.

[58] Z. Zhang, B. He, and K. Chang, “Understanding web query interfaces: best-effort parsing with hidden syntax,” in *SIGMOD*, 2004.

[59] B. He, K. Chang, and J. Han, “Discovering complex matchings across web query interfaces: A correlation mining approach,” in *KDD*, 2004.

[60] E. Dragut, C. Yu, and W. Meng, “Meaningful labeling of integrated query interfaces,” in *VLDB*, 2006.

[61] B. He and K. Chang, “Statistical schema matching across web query interfaces,” in *SIGMOD*, 2003.