A Random Walk Approach to Sampling Hidden Databases

Arjun Dasgupta
University of Texas at Arlington
arjundasgupta@uta.edu

Gautam Das *
University of Texas at Arlington
gdas@uta.edu

Heikki Mannila
HIIT, Helsinki University of Technology and University of Helsinki
mannila@cs.helsinki.fi

ABSTRACT

A large part of the data on the World Wide Web is hidden behind form-like interfaces. These interfaces interact with a hidden back-end database to provide answers to user queries. Generating a uniform random sample of this hidden database by using only the publicly available interface gives us access to the underlying data distribution. In this paper, we propose a random walk scheme over the query space provided by the interface to sample such databases. We discuss variants where the query space is visualized as a fixed and random ordering of attributes. We also propose techniques to further improve the sample quality by using a probabilistic rejection based approach. We conduct extensive experiments to illustrate the accuracy and efficiency of our techniques.

Categories and Subject Descriptors
H.3.3 Information Search and Retrieval

General Terms
Algorithms, Design, Performance, Measurement

Keywords
Hidden databases, sampling, top-k interfaces, random walk

1. INTRODUCTION

A large portion of data available on the web is present in the so called "deep web". The deep web (or invisible web or hidden web) is the name given to pages on the World Wide Web that are not part of the *surface web* (pages indexed by common search engines). It consists of pages which are not linked to other pages (e.g., dynamic pages which are returned in response to a submitted query). The deep web is believed to contain 500 times more data than the surface web [5]. A major part of data present on the hidden web lies behind form like interfaces. These form based interfaces are based on a back end proprietary database and a limited top-k query interface. The query interface is generally represented as a web form that takes input from users and translates them into SQL queries. These queries are then presented to the proprietary database and the top-*k* results provided to the user on the browser. Many online resource locater services are based on this model. To illustrate this scenario let us consider the example of a generic restaurant finder service:

Example 1: *Consider a web based form which lets the user choose from a set of attributes* $\{A_1, A_2, A_3, ... A_m\}$ *where each attribute may be Boolean, categorical or numeric, e.g., cuisine, price, distance, etc. The user chooses desired values for one or more of the attributes presented* (*or ranges in the case of numeric attributes*)*. This results in a query that is executed against a back-end restaurant database where each tuple represents a restaurant. The top-k answers* (*according to a ranking function*) *returned from the database are then presented to the user.*

The principle problem that we consider in this paper is: *given such a restricted query interface, how can one efficiently obtain a uniform random sample of the backend database by only accessing the database via the public front end interface?*

Database Sampling is the process of randomly selecting tuples from a dataset. Database sampling has been used in the past to gather statistical information from databases. It has a wide range of applications for the owner of the database. Given complete and unrestricted access to the database, many methods have been developed for efficiently selecting a random sample of the tuples of the database [14, 17].

However, in today's world where most databases are present behind a proprietary curtain, there needs to be some way to obtain statistical information about the underlying data with all these restrictions in place. This information can be then used to obtain insight into the data. Statistics about *a third party database* can be used to obtain quality, freshness and size information inside web sources. It can be used to identify uniformity or biases of topics. A typical example is as follows: Consider a web meta-service which retrieves data on restaurants in a city. It fetches information from two or more web sources which provide data on restaurants matching certain requirements. This service gets a query from the user and then feeds it to the two or more participating restaurant search engines. The final result is a combined mix of restaurant data from these underlying sources. Knowledge of the underlying databases would allow the makers of our restaurant service to call the database with the best quality and data distribution (in relation to a specific query) preferentially over the other sources.

However, generating random samples from hidden databases presents significant challenges. The only view available into these databases is via the proprietary interface that allows only limited access – e.g., the owner of the database may place limits on the type of queries that can be posed, or may limit the number of tuples that can be returned, or even charge access costs, and so on. The traditional random sampling techniques that have been developed cannot be easily applied as we do not have full access to the underlying tables.

In this paper we initiate an investigation of this important problem. For simplicity, we consider mainly single-table databases with Boolean, categorical or numeric attributes, where

* Partly supported by a grant from Microsoft Research

the front end interface allows queries in which the user can specify values of ranges on a subset of the attributes, and the system returns a subset (top-*k*) of the matching tuples, either according to a ranking function or arbitrarily, where k is a small constant such as 10 or 100.

Our main result is an algorithm called HIDDEN-DB-SAMPLER, which is based on performing *random walks* over the space of queries, such that each execution of the algorithm returns a random tuple of the database. This algorithm needs to be run an appropriate number of times to collect a random sample of any desired size. Note that this process may repeat samples - i.e., we produce samples "with replacement". There are two main objectives that our algorithm seeks to achieve:

- ***Quality of the sample:*** Due to the restricted nature of the interface, it is challenging to produce samples that are truly uniform. Consequently, the task is to produce samples that have small *skew*, i.e., samples that deviate as little as possible from the uniform distribution.
- ***Efficiency of the sampling process:*** We measure efficiency of the sampling process by the number of queries that need to be executed via the interface in order to collect a sample of a desired size. The task is to design an efficient procedure that collects a sample of the desired size as efficiently as possible.

Our algorithm is designed to achieve both goals - it is very efficient, and produces samples with small skew. The algorithm is based on three main ideas: (a) *Early termination*: Often, a random walk may not lead to a tuple. To prevent wasted queries, our algorithm is designed to detect such events as early as possible and restart a fresh random walk; (b) *Ordering of attributes*: The ordering of attributes that guides the random walk crucially impacts quality as well as efficiency – we show that for Boolean databases a random ordering of attributes is preferable over any fixed order, whereas for categorical databases with large variance among the domain sizes of the attributes, a fixed ordering of attributes (from small domains to large domains) is preferable for reducing skew; (c) *Parameter to tradeoff skew versus efficiency*: Since sample quality and sampling efficiency are contradictory goals, our algorithm is equipped with a parameter that can be tuned to provide tradeoffs between skew and efficiency.

A major contribution of this paper is also a theoretical analysis of the quantitative impact of the above ideas on improving efficiency and reducing skew. We also describe a comprehensive set of experiments that demonstrate the effectiveness of our sampling approach.

The rest of this paper is organized as follows. In Section 2 we formally specify the problem and describe a simple but inefficient random-walk based strategy that forms the foundation of our eventual algorithm. Section 3 is devoted to the development of HIDDEN-DB-SAMPLER for the special case of Boolean databases. In Section 4 we extend the algorithm for other types of data as well as other query interfaces. Section 5 discusses related work, and Section 6 contains a detailed experimental evaluation of our proposed approach. We conclude in Section 7.

2. PRELIMINARIES

2.1 Problem Specification

Throughout this paper our discussion revolves around hidden databases and their public interfaces. We start by defining the simplest problem instance. Consider a database table D with n tuples $\{t_1, \ldots, t_n\}$ over a set of m attributes $A = \{A_1, \ldots, A_m\}$. Let us assume that the attributes are Boolean – later in Section 4 we extend this scenario such that the attributes may be categorical or numeric. We also assume that duplicates do not exist, i.e., no two tuples are identical. This hidden backend database is accessible to the users through a public web-based interface. We assume a prototypical interface, where users can query the database by specifying the values of a subset of attributes they are interested in. Such queries are translated into SQL queries with conjunctive selection conditions of the form "SELECT * FROM D WHERE X_1=x_1 AND … AND X_s=x_s", where each X_i is an attribute from A and x_i is either 0 or 1. The set of attributes $X = \{X_1, \ldots, X_s\} \subseteq A$ is known as the set of attributes *specified* by the query, while the set $Y = A - X$ is known as the set of *unspecified* attributes.

Let $Sel(Q) \subseteq \{t_1, \ldots, t_n\}$ be the set of tuples that satisfy Q. Most web query interfaces are designed such that if $Sel(Q)$ is very large, only the top-k tuples from the answer set are returned, where k is usually a fixed constant such as 10 or 100. The top-k tuples are selected by a ranking function, which is either specified by the user or defined by the system (e.g., in home search websites, a popular ranking function is to order the matching homes by price). In some applications, there may not even be any ranking function, and the system simply returns an arbitrary set of k tuples from $Sel(Q)$. These scenarios, where the answer set cannot be returned in its entirety, are called *overflows.* At the other extreme, if the system returns no results (e.g., the query is too specific) an *underflow* occurs. In all other cases, where the system returns k or less tuples, we have a *valid* query result.

For the purpose of this paper, we assume that when an overflow occurs, the user cannot get complete access to $Sel(Q)$ simply by "scrolling through the rest of the answer list". The user only gets to see the top-k results, and the website also notifies the user that there was an overflow. The user will then have to pose a new query, perhaps by reformulating the original with some additional conditions. For most of the paper, for ease of exposition, we restrict our attention to the case when the front end interface restricts $k = 1$. I.e., for each query, either there is an overflow, or an underflow, or a single valid tuple is returned. The case of $k > 1$ is discussed in Section 4.

The principal problem that we consider in this paper is: *given such a restricted query interface, how can one efficiently obtain a uniform random sample of the backend database by only accessing the database via the front end interface?* Essentially the task is to develop a "hidden database sampler" procedure, which when executed retrieves a random tuple from D. Thus, such a sampler can be repeatedly executed an appropriate number of times to get a random sample of any desired size.

Of course, since accessing the tuples of a hidden database uniformly at random is difficult, such database samplers may not be able to achieve perfect uniformity in the tuple selection process. To be able to measure the "quality" of the random samples obtained, we need to measure how deviant is the sample distribution from the uniform distribution. More precisely, let the $p(t)$ be the *selection probability* of t, i.e., probability that the sampler selects tuple t from the database when executed once. We define the skew of the sample distribution as the standard deviation of these probabilities, i.e.,

$$skew = \sqrt{\frac{\sum_{1 \le i \le n} (p(t_i) - 1/n)^2}{n}}$$

Next, we define the notion of *efficiency* of the random sampler. We will measure efficiency by simply counting the total number of queries posed by the sampler to the front end interface in order to get a random sample of a desired size. Clearly, this notion of efficiency is rather simplistic – for instance it assumes that all queries take the same time/cost to execute. However, it is instructive to investigate sampling performance even with this simplistic measure – which we do in this paper, and leave more robust efficiency measures for future work.

Thus, our problem reduces to obtaining a random sample of a desired size with the least skew efficiently. As we shall see later, small skew and efficiency are conflicting goals – a very efficient sampler is likely to produce highly skewed samples and vice versa. Indeed, as we shall see later, the samplers that we design do exhibit these tradeoffs. In fact, the samplers that we develop have parameters that can smoothly tradeoff skew against efficiency.

2.2 Random Walks through Query Space

In this sub-section, we develop a simple approach to sampling hidden databases. This approach is naïve and inefficient, but it provides the foundation for the actual algorithm.

Brute Force Sampler: One extremely simple algorithm that will produce perfect uniform random samples is the BRUTE-FORCE-SAMPLER which does the following. Generate a random Boolean tuple of m-bit, and query the interface to determine whether such a tuple exists. I.e., the query will be a complete specification of all the tuple values, for which there are two possible outcomes: either the query underflows, or else it returns a valid result. The sampler repeats these randomly generate queries until a tuple is returned.

	A_1	**A_2**	**A_3**
t_1	0	0	1
t_2	0	1	0
t_3	0	1	1
t_4	1	1	0

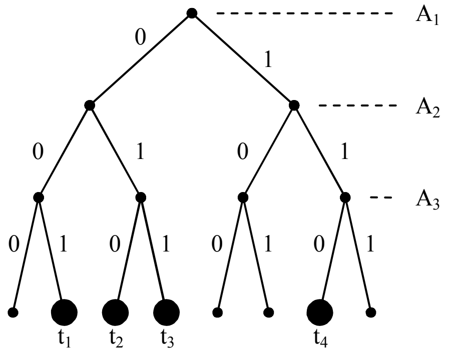

Figure 1: Random walk through query space

It is easy to see that this process will produce a perfect uniform random sample. However, this is an extremely inefficient process, especially when we realize that the size of most databases is much smaller that the size of the space of all possible tuples (i.e., $n << 2^m$). Thus the *success probability* of BRUTE-FORCE-SAMPLER, i.e., the probability of reaching a valid tuple, is $n/2^m$.To get a desired sample size of r, the expected number of queries that will be executed is $r*(2^m/n)$.

Random Walk View of Brute Force Sampler: An alternative view of this simple algorithm – which lays the foundations for the more sophisticated algorithm later in the paper – is to imagine a random walk through a binary tree in which the database tuples exist at some of the leaves. To make this more precise, assume a specific ordering of all attributes, e.g. $[A_1A_2…A_m]$. Consider Figure 1 which shows a database with three attributes and four tuples, and a complete binary tree with 4 (= m+1) levels, where the i^{th} level ($i \le m$) represents attribute A_i and the leaves represent possible tuples. The left (resp. right) edge leading out of any internal node is labeled 0 (resp. 1). Thus, each path from root to a leaf represents a specific assignment of Boolean values to attributes. Thus the leaves represent combinations of all possible assignments of values to the attributes. Note that only some of the leaves correspond to actual tuples in the database. In a real-world database only a small proportion of the leaves will correspond to actual tuples, and vast majority of the remainder will be empty.

The brute force sampler may be viewed as executing a random walk in this tree. We start with the first attribute A_1 and pick either 0 or 1 with equal probability. Next we pick either 0 or 1 as A_2's value and continue this walk till we reach a leaf. This random walk is essentially a random assignment of values to all attributes – i.e., it corresponds to the generation of a random query in the brute force sampler described above. This randomly generated query is then fed to the interface, which then either answers with a valid query, or fails (i.e. underflows).

2.3 Table of Notations

Table 1 lists all the notations that are used throughout the paper (some of these concepts will be introduced later in the paper).

Table 1: Notations used in paper

Notation	**Semantics**
n	Number of tuples in the database
m	Number of attributes in the database
$Sel(Q)$	Answer set of query Q, i.e., tuples that satisfy selection condition of query
$A_1…A_m$	Attributes of the database
$p(t)$	*Selection probability*, i.e. the probability with which tuple t gets selected by a random sampler
$s(t)$	*Access probability*, i.e., the probability with which a tuple is reached via a random walk
$a(t)$	*Acceptance probability*, i.e., the probability with which a tuple gets accepted into the sample, once it has been reached by a random walk.
$d(t)$	*Depth* i.e. length of the shortest prefix of the path that leads to t such that the corresponding query returns the singleton tuple t
F	*Failure probability*, i.e. the probability that a random walk leads to an underflow and has to be aborted
S	*Success probability*, = 1 - F
C	*Scaling factor* used to boost acceptance probabilities of all *tuples*

3. RANDOM WALK BASED SAMPLING

In this section we develop the main ideas behind HIDDEN-DB-SAMPLER, our algorithm for sampling the tuples of a database that is hidden behind a proprietary front end query interface. In this section we assume Boolean databases only.

3.1 Improving Efficiency – Early Detection of Underflows and Valid Tuples

We propose the following modification to BRUTE-FORCE-SAMPLER that significantly improves its efficiency. Assume that we have selected a fixed ordering of the attributes. Instead of taking the random walk all the way until we reach a leaf *and then* making a single query, what if we make queries *while* we are coming down the path? To make this more precise, suppose we have reached the ith level and the path thus far is $A_1=x_1$; $A_2=x_2 \ldots A_{i-1}=x_{i-1}$. Before proceeding further, we can execute the query that corresponds to this prefix of the walk. If the outcome is an underflow, we can immediately abort the random walk. If the outcome is a single valid tuple, we can select that tuple into that sample. And only if the outcome is an overflow do we proceed further down the tree. This situation is described in Figure 1. Note that if the algorithm proceeded along the path [00], it will detect the valid tuple t_1 and stop. Similarly, if it proceeded along the path [1], it will detect the valid tuple t_4 and stop. However, to detect the valid tuple t_3, it has to proceed along the path [011].

We discuss the impact of this proposed modification to BRUTE-FORCE-SAMPLER. Consider a tuple t in the database. Define the *depth* $d(t)$ of t to be the length of the shortest prefix of the path that leads to t such that the corresponding query returns the singleton tuple t. Thus for the example in Figure 1, $d(t_1)=2$, $d(t_2)=3$, $d(t_3)=3$ and $d(t_4)=1$. For certain databases and for certain ordering of attributes, the following significant improvements can occur:

- The average value of $d(t)$ can be substantially smaller than m, i.e., we will rarely have to go all the way to the leaves. Likewise, the random walks that lead to underflows can be fairly short.
- Moreover, the *success probability* (S) of a random walk leading to a valid tuple is substantially larger than the brute force sampler.

For the example in Figure 1, the paths [00], [010], [011] and [1] lead to valid tuples t_1, t_2, t_3 and t_4 respectively, whereas *no paths* lead to failure. Thus the success probability is 1.

Of course, one has to remember that in the brute force sampler, each random walk is associated with only one query that is executed at the end of the walk, whereas in the early detection approach we have to execute queries when visiting each node during a random walk. In spite of this, the aggregated number of queries executed (for obtaining the same sample size) in the latter is substantially smaller.

We now provide some theoretical justification as to why the success probability may be substantially larger than that of the brute force sampler. Our theoretical results are limited to certain simple i.i.d generated datasets (deriving success/failure probabilities for arbitrary datasets appears to be substantially harder), but nevertheless they provide the inspiration for our proposed algorithmic modification. Moreover, our experiments on a variety of real as well as synthetic datasets (Section 6) also reinforce the advantage of this early detection approach.

Consider a dataset D with n tuples having i.i.d. binary attributes with the probability of a 1 being p. Assume any arbitrary ordering of attributes, and consider a random walk starting from the root of D. Let $F(n, p)$ be the probability of a *failure* in the walk. We have the following boundary conditions and recurrence for $F(n, p)$:

Theorem 1: *Given an i.i.d. Boolean dataset with probability of 1's being p, and any ordering of attributes, we have*

$$F(1, p) = 0$$
$$F(0, p) = 1$$
$$F(n, p) = \sum_{i=0}^{n} \binom{n}{i} p^{i} (1-p)^{n-i} F(i, p)$$

Clearly, in a tree with one tuple the query will return the tuple, and success is certain, while in a tree with no tuples failure is certain. For a tree with n rows with attribute values independent and identically distributed, with the probability of a 1 being p, the right branch (corresponding to the value 1) will have i nodes with the binomial probability $\binom{n}{i} p^{i}(1-p)^{n-i}$, and the failure probability in the walk going to the right branch is $F(i, p)$. This concludes the proof of the theorem.

Note that if we interchange the values for the boundary conditions, we get an expression for the *success* of a random walk. Figure 2 shows a MATLAB simulation of $F(n, p)$ as a function of n for various values of p. We observe that the failure probability is the smallest for $p = 0.5$, but even for other values of p the probability of success is reasonably high. The convergence of the curves for $F(n, p)$ are to be expected: the third equation in Theorem 1 is satisfied by any function $F(n, p)$ that is constant with respect to n.

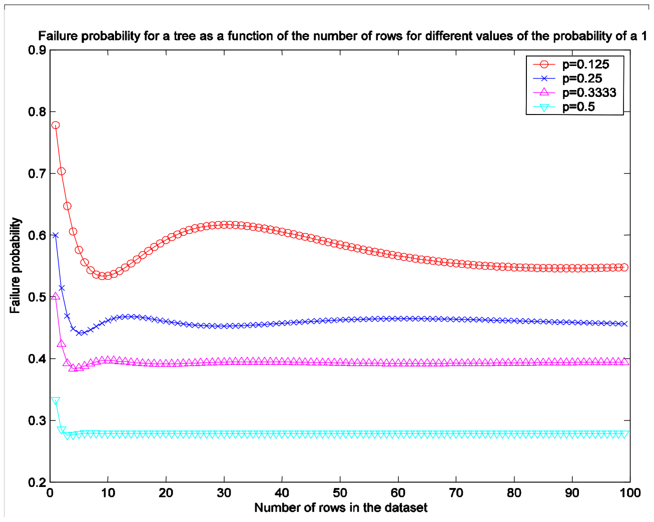

Figure 2: Failure probability in i.i.d. databases as a function of the number of tuples

In contrast, note that the success probability of the brute force sampler ($n/2^m$) is significantly smaller because it depends upon m. Thus, early detection of underflows is crucial in increasing the efficiency of the sampler.

However, this increased efficiency comes at a price, as *skew* is introduced in the sample distribution. We discuss the issue of skew next, and how it can be controlled.

Aside: There is one further issue that merits discussion. While a random walk is in progress and queries are leading to overflow, legitimate database tuples are being returned via the query interface (recall that for each overflowing query, the system returns *k* tuples from *Sel*(*Q*)). However, these tuples are useless for assembling into a random sample because they have not been selected by a random procedure; they are either the top-*k* tuples with the highest scores (in case a ranking function is used), or may even have been arbitrarily picked by the system. Consequently they have to be ignored and the walk has to continue.

3.2 Reducing Skew – Random Ordering of Attributes

Early detection of underflows introduces skew into the sample. To see this, recall from Table 1 that that $s(t)$ is the *access probability* of tuple *t*, i.e., the probability that tuple *t* is reached by a random walk (thus the *selection probability*, $p(t) = s(t)/S$ where *S* is the success probability of a random walk reaching any valid tuple). It is easy to see that $s(t) = 1/2^{d(t)}$. But since there is variance among the depths at which the database tuples are detected, there is variance among the values of $s(t)$, which contributes to the skew. The skew depends upon the specific database and the specific ordering of attributes used. For the example in Figure 1, $s(t_1) = 1/4$, $s(t_2) = 1/8$, $s(t_3)=1/8$ and $s(t_4)=1/2$.

We first provide theoretical analysis that quantifies the skew for certain i.i.d. databases (as before, we point out that analytical derivations of skew for general databases appears to be extremely difficult). We follow this analysis with a discussion of techniques for reducing skew without adversely impacting efficiency.

Quantifying Skew: Let *D* be an i.i.d. 0-1 database with *n* rows where the probability of a 1 is 0.5. Assume any ordering of the attributes. For a tuple *t*, we shall refer to $t(1\ldots x)$ as the prefix corresponding to the first *x* values of the tuple according to this order. Since skew is defined as the standard deviation of $p(t)$, and $p(t) = s(t)/S$, we basically need to analytically derive the standard deviation of s(t).

Theorem 2: *Given an i.i.d. Boolean dataset with p=0.5 and any ordering of attributes, we have*

$$E[s] \approx \frac{1}{2n \ln 2}$$

$$Var(s) \approx \frac{3}{4 \ln 2(n^2+n)} - \frac{1}{4n^2(\ln 2)^2}$$

We prove this theorem and at the same time provide some intuition for the distribution of the probabilities Consider the distribution of the access probability of rows in the i.i.d. model. Consider a row *t* in the database. Recall that the *depth d(t)* of t is the length *x* of the shortest prefix $t(1\ldots x)$ of *t* such that the query corresponding to this prefix returns the singleton tuple *t*. Note that given the database *D* and an ordering of the attributes, the depth $d(t)$ is a fixed quantity. We analyze next the distribution of the depths of rows in the tree. Let *q(x)* be the probability that a given tuple *t* in *D* has depth at most *x*. This happens if each of the other tuples *u* in *D* differs from *t* in at least one of the positions $1\ldots x$, i.e., if for each *u* it is not the case that *u* agrees with *t* on the *x* first positions, i.e., with probability $(1-2^{-x})$. As the tuples are independent, the probability that this happens for each of the *n* - 1 other rows in *D* is $q(x) = (1-2^{-x})^{n-1}$. Thus the probability $r(x)$ that the depth of a tuple *t* is exactly *x* is

$$r(x) = q(x) - q(x-1) = (1-2^{-x})^{n-1} - (1-2^{-x+1})^{n-1}$$

Figure 3 shows a MATLAB simulation of the distribution of $r(x)$ for different values of *n*. We see that while the depths of the nodes are strongly concentrated around log(*n*), there still are a considerable number of rows for which $d(t)$ differs a lot from the mean. As $s(t) = 2^{-d(t)}$, such differences translate to large differences in the sampling probabilities.

The access probability $s(t) = 2^{-d(t)}$ is the probability of producing a query string that agrees with *t* for $d(t)$ positions.

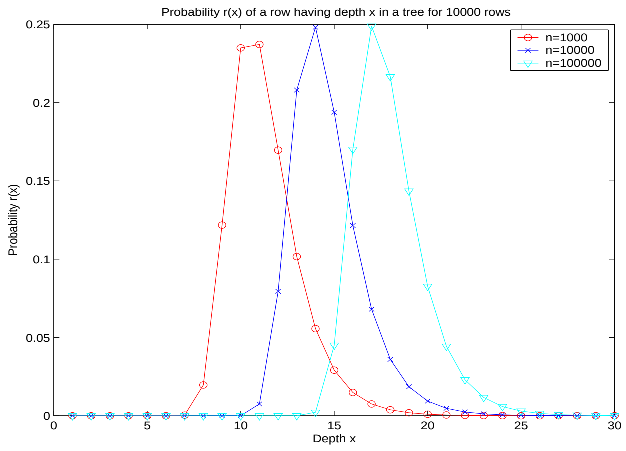

Figure 3: The probability $r(x)$ of a random row having depth *x* in an i.i.d. database with *n* rows where p = 0.5

Thus the probability that a tuple is selected depends very strongly on its depth, and the sampling procedure has a strong bias towards rows with small depth, i.e., towards tuples that are separated from others by short query prefixes.

We now leverage the relationship between the aggregate of the access probabilities and the failure probability of the database. It is easy to see that

$$\sum_{t \in D} s(t) = 1 - F(n,p)$$

This is because the probability that the walk terminates one of the valid tuples is exactly 1 - $F(n, p)$. Thus

$$\sum_{t \in D} s(t) = 1 - F(n,p) = \sum_{x=0}^{\infty} n\, r(x) 2^{-x} \qquad (1)$$

$$= \sum_{x=0}^{\infty} n(q(x) - q(x-1)) 2^{-x}$$

This gives a non-recursive expression for $F(n, p)$. The values computed from (1) correspond exactly to the values obtained from Theorem 1. An asymptotic analysis of (1) can be done by replacing sums with integrals; we thus obtain

$$\sum_{t \in D} s(t) = \sum_{x=1}^{\infty} n r(x) 2^{-x}$$

$$\approx \int_1^{\infty} n(q(x) - q(x-1)) 2^{-x} dx \approx (2 \ln 2)^{-1} \approx 0.72.$$

This is in good agreement with Figure 2. Thus the expected value of the access probability is

$$E[s] = (1/n)\sum_{x=1}^{\infty} nr(x)2^{-x} \approx (2n \ln 2)^{-1}$$

The variance of $s(t)$ can also be evaluated. After some simplifications we obtain

$$E[s^2] = \sum_{x=1}^{\infty} nr(x)2^{-2x}$$
$$\approx \int_1^{\infty} (q(x) - q(x-1))\, 2^{-2x}\, dx = \frac{3}{4 \ln 2}(n^2 + n)^{-1}$$

and

$$Var[s] = E[s^2] - E[s]^2 \approx \frac{3}{4 \ln 2(n^2+n)} - \frac{1}{4n^2(\ln 2)^2}$$

This concludes the proof of Theorem 2. The analysis shows that for i.i.d. data with p=0.5, and for any ordering of attributes, the deviation *Std*[s] of s is very close to $E[s]$. As a MATLAB simulation shows in Figure 4, the "relative" skew is about 1.

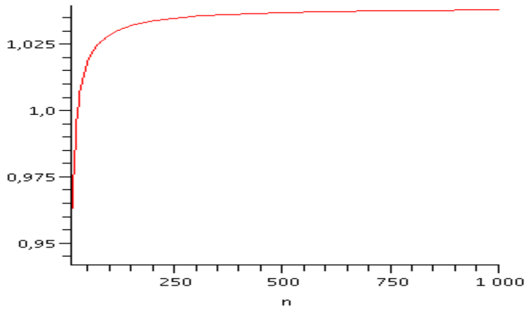

Figure 4: The ratio *Std*[*s*]/*E*[s] as a function of the number of nodes.

Although Theorem 2 has been derived for specific i.i.d databases, the flavour of the result has been corroborated by our experiments on a variety of datasets. Samples collected by the random sampler modified with early underflow detection exhibit skew. Therefore the task ahead of us is to develop additional techniques that reduce the skew and yet do not adversely impact efficiency. We discuss these techniques next.

Reducing Skew by Random Ordering of Attributes: In our efforts to reduce skew, we observe the importance of a having a favorable ordering of attributes that reduces the variance of $s(t)$ (or equivalently, of $p(t)$). If the variance in the length of each walk is small, the skew will also be small. A very simple approach is to preface each random walk with a *random ordering* of the attributes, and use the resultant ordering to direct the random walk. In our case, the intuition behind random orderings of attributes is as follows. For a fixed ordering, the depth of a tuple, $d(t)$, is fixed. If $d(t)$ is large, then $s(t)$ is small, and therefore t is less likely to be reached by a random walk, whereas if $d(t)$ is small, then $s(t)$ is large, and t is more likely to be reached. With random orderings of attributes, the probability $s(t)$ for a tuple t now becomes a random variable whose expected value is much closer to the average value of the access probabilities of all tuples.

In fact, as we theoretically analyze below, it can be shown for i.i.d. datasets with p=0.5, if we employ random orderings before initiating random walks, there is no skew in the sampling process. We caution that this theoretical result does not extend to more general datasets (e.g., when p is different from 0.5, or correlated attributes), and thus only serves as supporting evidence for adopting random orderings in our sampler. However our experiments (Section 6) on a variety of real and synthetic databases make it clear that random reordering does indeed reduce skew, sometimes dramatically so, without any appreciable decrease in performance.

Consider a i.i.d Boolean dataset with p=0.5. Now consider the sampling algorithm in the case when the attributes are randomly reordered for each random walk. In this case the depth $d(t)$ of a row t is a random variable, and the access probability $s(t)$ for t is obtained by two randomizations: the first is the selection of the random ordering of the attributes, and the second is the random walk determined by the random selection of values in the query.

Theorem 3: *Given an i.i.d. Boolean dataset with p=0.5 and a random sampler with random reordering of attributes as well as early termination, the resulting skew = 0.*

The key observation is the following. As in Theorem 2, denote by $r(x)$ the probability that a random tuple has depth x in an i.i.d. database; earlier we derived a simple expression for $r(x)$. Let then $r'(x)$ be the probability that a fixed tuple has depth x, when the probability is taken over random re-orderings of the attributes. We have $r(x) = r'(x)$ by the i.i.d. property: a fixed tuple t has depth in a random reordering of the attributes exactly with probability $r(x)$. (I.e., as functions from {0, 1, 2,…} to nonnegative reals, the functions r and r' are identical.).

In a random walk with a fixed ordering of the attributes the probability that a fixed tuple t will be reached is $2^{-d(t)}$, which for most tuples is different from the uniform value of $(1-F(n,p))/n$. However, in a randomly reordered run the probability that the fixed tuple t is reached is

$$\sum_{x=0}^{\infty} r'(x)2^{-x} = \sum_{x=0}^{\infty} r(x)2^{-x} = (1 - F(n,p))/n,$$

i.e., the randomly reordered random walk produces un-skewed results. This concludes the proof of Theorem 3.

The result that random reordering leads to un-skewed sampling in the i.i.d. case does not hold for arbitrary databases. For example, suppose that there is a specific tuple t such that for half of the attributes t is the only tuple having a 1 in that attribute. Then any random ordering will, with high probability, include one of these attributes early on, and thus there is an overwhelming bias towards selecting row t. Thus, in general, we need the acceptance/rejection method that will be discussed next.

3.3 Reducing Skew – Acceptance/Rejection Sampling

In this section we consider another idea that serves to reduce skew. So far, we had assumed that whenever a tuple is reached via a random walk, it is accepted into the sample. But we know that the probability of reaching a tuple varies from tuple to tuple, depending on the depth at which the tuple is uniquely identified.

We know that skew is the result of variance among the access probabilities. To counter this skew, we propose *rejection sampling*, a procedure by which tuples are probabilistically accepted or rejected once they have been reached by the random walk. In other words, rejection sampling is used to compensate for the deviation in the access probabilities.

We make this idea precise as follows. Consider tuple t that is reached by a random walk. For that specific order of attributes, its access probability is $s(t) = 1/2^{d(t)}$. Let us define $a(t)$ as the *acceptance probability* for tuple t, i.e., once t is reached, it is accepted with probability $a(t)$. Thus, the overall probability of selecting tuple t (for that specific ordering of attributes) is $s(t) \times a(t) = a(t) / 2^{d(t)}$

So what is an appropriate value for $a(t)$? Since our goal is to make the probability of selecting a tuple the same for all tuples (i.e., skew = 0), this can be achieved if we make $a(t)$ proportional to $2^{d(t)}$. However, since $a(t)$ is a probability, we have to ensure that it is between 0 and 1.

One seemingly reasonable setting is $a(t)=2^{d(t)}/2^m$, which is guaranteed to be between 0 and 1 since $1 \leq d(t) \leq m$. This way, we can ensure that the probability of selecting t is $1/2^m$, i.e., the same for all tuples. Clearly, we will now be able to produce a sample without any skew, since this probability is the same for all tuples. However, we have reintroduced inefficiency into the sampler, since even though our random walks stop early, most of the time the destination tuple gets rejected, leading to wasted walks.

Fortunately, it is not necessary to set the acceptance probabilities to be that small. Consider Figure 5 which shows a binary tree for a specific ordering of attributes. Notice that tuples are reached or uniquely identified at different depths.

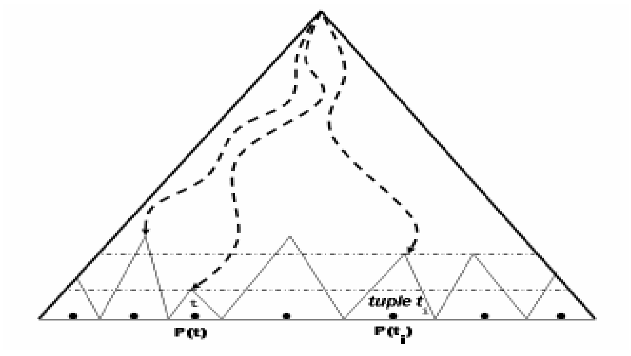

Figure 5: Different Depths

Suppose we knew $d_{\max}$, the largest value of $d(t)$ over all possible trees (corresponding to all possible attribute orders) and all tuples. It is easy to see that setting $a(t) = 2^{d(t)}/2^{d\max}$ would also produce unbiased samples, but possibly more efficiently than described above, in case $d_{\max}$ is smaller than m.

However, $d_{\max}$ may still be very large, rendering the approach inefficient. Moreover, it is unrealistic to assume that $d_{\max}$ is known (or can be easily computed) beforehand. To overcome these problems, we adopt the approach described next.

Boosting Acceptance Probabilities by Scaling Factor

We adopt the compromise approach where we boost the acceptance probabilities of each tuple by a *Scaling Factor C*. Let C be a constant ≥>= $1/2^m$. We define $a(t)$ as

$$a(t) = \min\{ C 2^{d(t)}, 1\}$$

Let us discuss the impact of C on the sampling process. If C is $\leq 1/2^{d\max}$ then we would still have un-skewed samples. However, if C is greater than $1/2^{d\max}$, then there is a chance that some tuples that get identified after very long walks will get accepted with probability 1 (the *min* operator is required in the definition of $a(t)$ above to guarantee that it remains a probability), thus introducing skew into the sample. Larger the C, more the chances of such tuples entering the sample and thereby increasing skew. On the other hand, a large skew increases efficiency, as the acceptance probabilities are boosted by C. Thus we may regard C as a convenient parameter that provides tradeoffs between skew and efficiency.

How do we estimate a suitable value for C? Recall that random ordering of attributes is the primary mechanism for reducing variance among the access probabilities. Based on our experimental evidence, it appears that setting C to be $1/2^{d'}$, where d' is somewhat smaller than the the average depth at which tuples get uniquely identified, will work well. This can be done adaptively where the average tuple depth is learned as more and more random walks are accomplished.

3.4 Algorithm HIDDEN-DB-SAMPLER

In summary, we have suggested three ideas that can improve the performance of BRUTE-FORCE-SAMPLER, and which we adopt in our HIDDEN-DB-SAMPLER: These ideas are (a) early detections of underflow and valid tuples, (b) random reordering of attributes, and (c) boosting acceptance probabilities via a scaling factor C. Thus, three random procedures must be followed, in sequence, before a tuple get accepted into the sample.

Figure 6 gives the pseudo-code of HIDDEN-DB-SAMPLER for Boolean databases.

```
Algorithm HIDDEN-DB-SAMPLER
          for Boolean databases

 1. Generate random permutation [$A_1 A_2 … A_m$]
 2. $Q$ = {}
 3. for $i$ = 1 to m
 4.      $x_i$ = random bit 0 or 1
 5.      $Q$ = $Q$ AND ($A_i$ = $x_i$)
 6.      Execute $Q$
 7.      if underflow
 8.           go to 1
 9.      else if overflow
10.           continue
11.     else  {    $t$ = Sel($Q$)
                      //when $k$=1, Sel($Q$)
                      // has one tuple
12.               Toss coin with bias
                      min{$C*2^i$, 1}
13.               if head return t
14.               else start again from 1 }
```

Figure 6: Algorithm for random sampling from hidden Boolean databases

4. EXTENSIONS

The algorithm we developed in Section 3 was for the simplest scenario, where the database was Boolean and the number of returned tuples was at most 1. In this section we extend the algorithm to work for more general scenarios.

4.1 Generalizing for k > 1

Most front end interfaces return more than one tuple, and *k* is usually in the range of 10-100. It is fairly straightforward to extend HIDDEN-DB-SAMPLER for more general values of *k*. In fact, when *k* is large, the efficiency of the algorithm actually increases.

Essentially, the algorithm is the same as before, but the random walk terminates either when there is an underflow, or when a valid result set is returned (say $k' \leq k$ tuples). One can see that in the latter case the termination is at least log(*k'*) levels higher. Once these *k'* tuples are returned, the algorithm picks one of the *k'* tuples with probability 1/*k'*. I.e., the access probability of the tuple that gets picked is therefore

$$s(t) = \frac{1}{k' 2^{d(t)-1}}$$

Then, the tuple is accepted with probability *a*(*t*) where $a(t) = \min\{ Ck' 2^{d(t)-1}, 1\}$

where *C* is a scale factor that boosts the selection probabilities to make the algorithm efficient.

4.2 Categorical Databases

We define a categorical database to be one where each attribute A_i can take one of several values from a multi-valued categorical domain Dom_i. Many real-world databases have categorical attributes, and most front end interfaces reveal the domains of each attribute – usually via drop down lists in query forms. Most of the algorithmic ideas remain the same as for the Boolean database, the only difference being that the fan out at a node at the *i*th level of the tree is equal to the size of Dom_i. The random walk selects one of the $|Dom_i|$ edges at random. The access probability for tuple *t* is therefore defined as

$$s(t) = \frac{1}{\prod_{1 \leq i \leq d(t)} |Dom_i|}$$

The rest of the extensions to the algorithm are straightforward. However, a crucial point is worth discussing here. If the domain sizes of the attributes are more or less the same, then the random ordering of attributes plays an important role in reducing skew. However, if there is large variance among the domain sizes – e.g., in a restaurants database for a city, there may be attributes with large domains such as "zipcode", along with a attributes with small domains such as "cuisine" – the fixed order of sorting the attributes from smallest to largest domains produces smaller skew compared to random orderings. This is because most of the small domain values have numerous representative tuples, and hence for this fixed order most of the walks proceed almost to the bottom of the tree before the walk can uniquely identify a tuple. Hence the variation in depth is small. In contrast, if a large domain attribute such as zipcode appears near the top of the tree in a random order, the walk will quite early encounter subtrees that contain few tuples. Thus some of the walks will terminate much faster and the depths will have more variance. Likewise, the inverse argument says that ordering the attributes from largest to smallest domains will be more efficient but will produce larger skew. In our experiments (Section 6) we include some results involving real-world categorical databases.

4.3 Numerical Databases

Real-world databases often have numerical attributes. Most query interfaces reveal the domain of such attributes and allow users to specify numeric ranges that they desire (e.g., a "Price" column in a homes database may restrict users to specifying price ranges between $0 and $1M).

If we can partition each numeric domain into suitable discrete ranges, our random sampler can work for such databases by treating each discrete range as a categorical value. However, there is a subtle problem that must be overcome: the discretization should be such that that each tuple in the database is unique in the resulting categorical version of the database. If the discretization is too coarse such that more than *k* tuples have identical representations in the categorical version, then we cannot guarantee a random sample because some of these tuples may be permanently hidden from users by an adversarial interface.

An alternate approach is to not discretize numeric columns in advance, but to repeatedly keep narrowing the specified range in the query during the random walk. We make this more precise as follows. For simplicity, consider a one-column database that has just one numeric attribute, *A*. We can divide the domain of A into two halves, randomly select one of the halves, and pose a query using this range. If there is an overflow, we can further divide this range into two, and select one at random to query again. This way we will eventually find a valid tuple. This approach can be easily extended to a multiple numeric attributes as follows. While the walk is progressing, a random attribute is selected, including attributes already selected earlier. If we select an attribute selected earlier, then we split its most recent queried range into two and pick one of the halves at random to query.

This approach has the following drawback. Note that if the numeric data distribution is spatially skewed (e.g, for a numeric attribute with domain [0, 1], the value of one tuple may be 0, and the values of the remaining n-1 tuples may be 1, 1- ε, 1 – 2ε …), then the first tuple will be selected for the sample with much higher probability than the remaining tuples. One way of compensating for this effect is to dynamically partition the most recent queried range into several ranges instead of just two halves, and then pick one of the ranges at random. In general, an interesting open problem is how many queries in this query model are needed to obtain an approximation to the probability density function of a real-valued attribute.

4.4 Interfaces that Return Result Counts

Some query interfaces return to the user the top-*k* results, and in addition the total count of all tuples that satisfy the query condition, |*Sel*(*Q*)|. E.g., most web search engines will return the top-*k* web pages, but also the size of the result set. We show how random sampling via such an interface can be done optimally without introducing skew. For simplicity, assume a Boolean database. Assume any ordering of the attributes. For each node *u* of the tree, let *n*(*u*) represent the number of leaves in the sub-tree rooted at *u* that correspond to actual tuples. In this scenario, a *weighted random walk* can be performed which is guaranteed to reach an actual leaf tuples on every attempt. Starting from the root, and at every node *u*, we select either the left or right branch with probability *n*(*left*(*u*))/*n*(*u*) and *n*(*right*(*u*))/*n*(*u*) respectively. These cardinalities can be retrieved by making appropriate queries via front end interface, At every node, edges are given weights to represent the density of their underlying subtree. Moreover,

$n(left(u))/n(u) + n(right(x))/n(x) = 1$. It is thus not hard to see that the selection probability of each tuple is $1/n$, thus guaranteeing no skew.

4.5 Interfaces that only Allow "Positive" Values to be Specified

Some web query interfaces only allow the user to select "positive" Boolean values. For example, a database of homes typically has a set of Boolean attributes such as "Swimming Pool", "3-Car Garage" and so on, and the user can only select a subset of these features to query. Thus, there is no way a user can request for houses that *do not* have swimming pools be retrieved.

While such interfaces are quite common, it is not always possible to collect a uniform random sample from such databases. Consider a tuple t_1 that "dominates" another tuple t_2, in the sense that for attributes that have value 1 in t_2, the corresponding attributes in t_1 are also 1. If k=1, then t_2 can be permanently hidden from the user by an adversarial interface that always prefers to return t_1 instead of t_2. The only way to solve this problem is to assume that no tuple dominates more than k other tuples.

5. RELATED WORK

Traditionally database sampling has been used to reduce the cost of retrieving data from a DBMS. Random sampling mechanisms have been studied in great detail e.g., [4, 8, 14, 15 and 17]. Applications of random sampling include estimation methodologies for histograms and approximate query processing using techniques (see tutorial in [6]).

However, these techniques do not apply to a scenario where there is an absence of direct access to the underlying database. A closely related area of sampling from a search engines index using a public interface has been addressed in [2] and more recently [1]. The technique proposed by [1], introduces the concept of a random walk on the documents on the World Wide Web using the top-k results from a search engine. However, this document based model is not directly applicable to hidden databases. In contrast to the database scenario, the document space is not available as a direct input in the web model. This leads to the use of estimation techniques which work on assumptions of uniformity of common words across documents. Random sampling techniques on graphs have been implemented using Markov Chain Monte Carlo techniques, e.g., Metropolis Hastings [12, 7] techniques and Acceptance/Rejection technique [13].

Hidden databases represent a major part of the World Wide Web and are commonly referred to as the hidden web. The size and nature of the hidden web has been addressed in [5], [9], [10] and [11]. Probing and classification techniques on textual hidden models have been addressed by [3] while techniques on crawling the hidden web were studied by [16].

6. EXPERIMENTATION AND RESULTS

In this section we describe our experimental setup, our results using the HIDDEN-DB-SAMPLER, and draw conclusions on the quality and performance of our technique. The results from a related model proposed by Bar-Yossef et. al. [1] for sampling web pages are also compared with our approach.

Hardware All experiments were run on a machine having 1.8 Ghz P4 processor with 1.5 GB RAM running Fedora Core 3. All our algorithms were implemented using C++ and Perl.

Implementation and Setup We ran our algorithm on two major groups of databases. The first group comprised of a set of small Boolean datasets with 500 tuples and 15 attributes each. (Small datasets makes it relatively easy to measure skew, because we can run the sampler millions of times, and measure the frequencies with which each tuple gets selected.) Several such datasets with varying proportions of 1 and 0 were synthetically generated. We defined p as the ratio of 1's in a Boolean set. For brevity, we have focused on i.i.d. data with p=0.5 and p=0.3 in addition to a mixed dataset with varying proportions of 1's (p=0.5, 0.3 and 0.2) combined vertically and horizontally with varied proportions.

A real dataset of 500 tuples manufactured using real data crawled from the Yahoo Autos website was also used. This was used to demonstrate the effectiveness of our methods for non-i.i.d. (i.e., correlated) data.

The other group comprised of large datasets of sizes between 300,000 to 500,000 each. These datasets were primarily used for performance experiments. Some of them were synthetically generated in a similar manner, with the exception of a real world dataset on restaurants information collected from www.yellowpages.com. The real dataset consisted of information on various restaurants with 10 attributes like price, cuisine, distance, ratings, etc having varying number of possible values (largest=25, smallest=5). All numerical attributes were grouped into finite ranges and categorized. To speed up execution, we simulated the interface and database retrieval by storing the data in main memory. The interface was used to look up top-k matches to a specific query. Different top-k values were also used to measure the change in performance. All execution and retrieval were done from main memory. These datasets with their associated interfaces were used to run experiments on the quality and performance of our technique.

Parameters We had defined skew as the standard deviation over the selection probabilities in Section 2.1. In all the experimental results described below, we actually discuss relative skew, which is skew times the size of the dataset. As discussed above, this measure is easily verifiable for small datasets. Unfortunately, it is not practical to measure skew for large datasets, because that would require running the sampler many times more than the size of the large datasets. Thus, for large datasets, we define a new variant of skew where we compared the marginal frequencies of a few selected attribute values in the original dataset and in the sample:

$$\textit{Marginal Skew} = \sqrt{\frac{\sum_{v \in V} \left(1 - \frac{p_S(v)}{p_D(v)}\right)}{|A|}},$$

Here V is a set of values with each attribute contributing a representative value, and $p_S(v)$ (resp. $p_D(v)$) is the relative frequency of value v in the sample (resp. dataset). The intuition is that if the sample is a true uniform random sample, then the relative frequencies of any value will be the same as in the original dataset. Quality measurements for a small number of samples generated from a large dataset were done using this measure. Efficiency (or performance) was measured by counting the number of queries that were executed by the sample to reach a certain desired sample size.

6.1 Quality of the samples

6.1.1 Small Databases

The HIDDEN-DB-SAMPLER was first run on the small datasets to generate a very large number of samples (around 300,000 samples were collected from the 500 tuple datasets). Two types of experiments were run here. First, we considered fixed ordering of the attributes in our datasets. We tried out several fixed orderings. Next, a random ordering was used for selecting attributes at every level in the walk. Next the idea of acceptance/rejection sampling was introduced. We begin by using the best value of the path that corresponds to the longest walk on the query space to accept tuples. Since a very large number walk are run for a small dataset, we could find the depth of the longest walk. The scaling factor *C* was then introduced. *C* was varied starting from the value corresponding to the longest walk and then progressively increased till we reached a point where the impact of *C* became ineffective and all tuples that were reached by the walks were getting accepted.

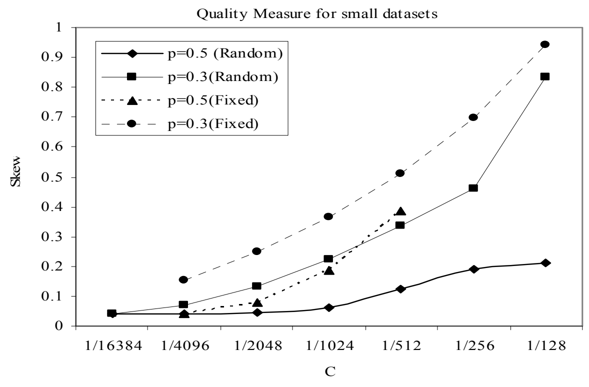

Figure 7: Effect of Scaling Constant C on Skew for small datasets

Figure 7 shows the results of these experiments. We observed that the random ordering of attributes does much better than any of the fixed orderings that we tried. This corroborates our previous analysis on the effectiveness of random ordering to improve quality. Moreover, for all cases as *C* was increased the value of skew got larger. This indicates the tradeoff in terms of skew while estimating *C*. Finally, the algorithm produced smaller skew in the case of p=0.5 compared to the dataset with p=0.3. (The theoretical skew for p=0.5 should be zero, but the non-zero skew that was observed was an artifact of the sampling process). Figures 8 and 9 shows similar behavior for mixed data and correlated data.

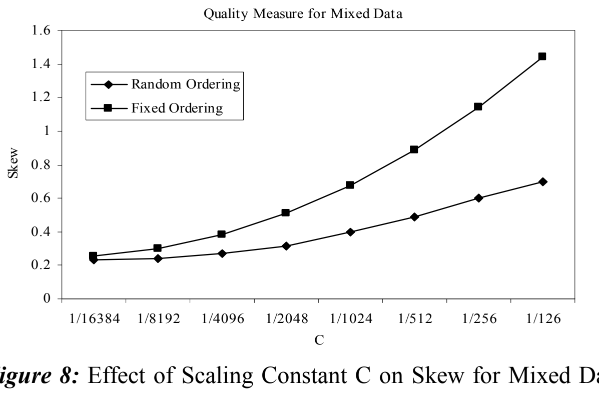

Figure 8: Effect of Scaling Constant C on Skew for Mixed Data

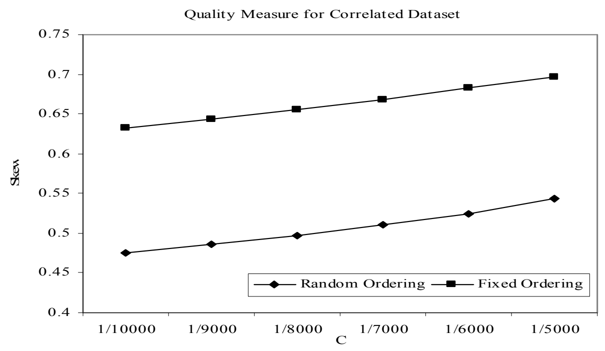

Figure 9: Effect of C on skew for Correlated data

6.1.2 Large Databases

We ran our experiments on synthetic large databases. In this scenario we limited our random walks to collect 5000 samples. Marginal Skew was used to measure quality. Marginal frequencies over all the “1” values of the attributes were collected in both the sample as well as the dataset. Figure 10 (*p*=0.3) represents the change in marginal skew for various values of the scaling constant *C*, for random ordering and a specific fixed ordering. As *C* increases, the marginal skew increases for both fixed and randomly ordered attributes with random ordering producing better results compared to any fixed ordering that we tried. We observe a similar trend as in small datasets. In contrast to these Boolean scenarios, the quality results from our experiments on the real world categorical database indicate lower marginal skews for the fixed order of attributes from (smallest domains to largest domains).

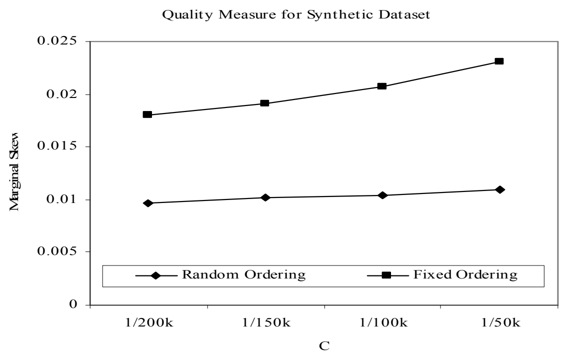

Figure 10: Marginal Skew v/s C for synthetic large data

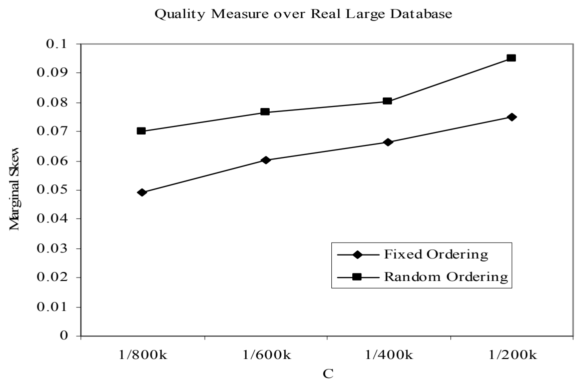

Figure 11: Marginal Skew based measure for real large data

In the categorical scenario, most of the attributes have small domains, and only very few have large domains. Thus, most of the walks proceed almost to the bottom of the tree since most of the small domain values have numerous representative tuples.

6.2 Performance of the Methods

6.2.1 Small Databases

We measure performance as the amount of work done, i.e., with the number of queries compared with the number of samples retrieved. This metric is used over all our performance experiments. Figure 12 indicates the queries versus sample size tradeoffs for a small mixed dataset. This experiment shows that the performance of fixed ordering is dependent on the specific ordering used (sometimes it performs better and sometimes worse *w.r.t.* random ordering).

Figure 13 indicates the increase in efficiency of our random order random walks when *k* in top-*k* is increased from a restrictive top-1 scenario. By increasing *k* we allow the random walks to terminate at a comparatively higher level in the query tree. Thus, performance improves as *k* is increased.

6.2.2 Large Databases

Figure 14 shows the number of queries v/s sample size for a 500,000 tuple database with *p*=0.3. As with the small datasets the performance of fixed order depends on the specific order used and was often worse than randomly ordered attributes. The situation for our categorical large dataset is however different (Figure 15). Performance is optimal for a fixed ordering of attributes in this scenario where the attributes with the largest domains are placed before the smaller ones. If the variance of attribute states is high as is in our case, the very first step of the random walk may reach sub-trees that contain very few tuples, and thus some of the walks will terminate much faster compared to other orders. Thus, when the variance in attribute states is high this type of an arrangement yields better performance.

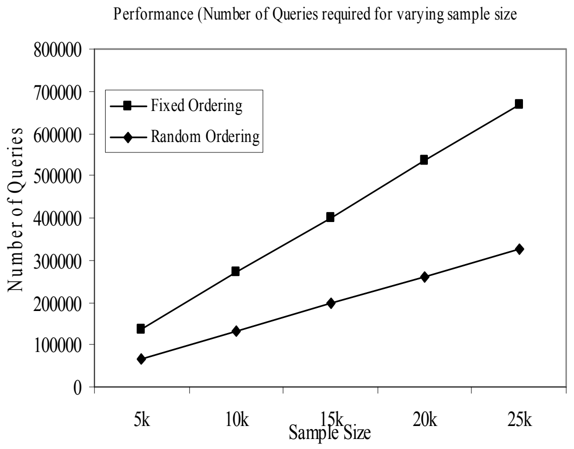

Figure 12: Performance of HIDDEN-DB-SAMPLER for a top-1 query interface

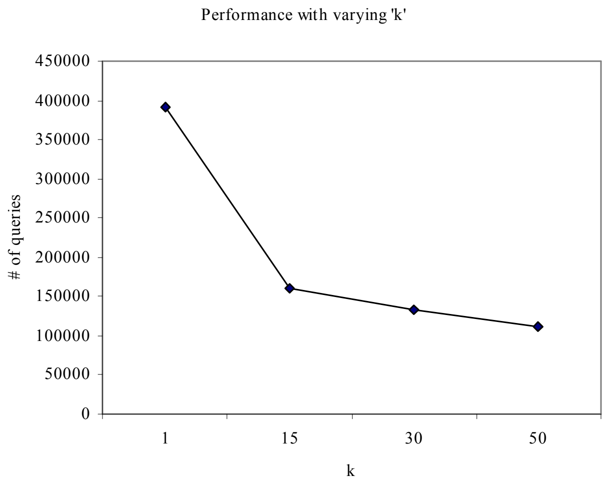

Figure 13: Effect of varying (top-) k on performance

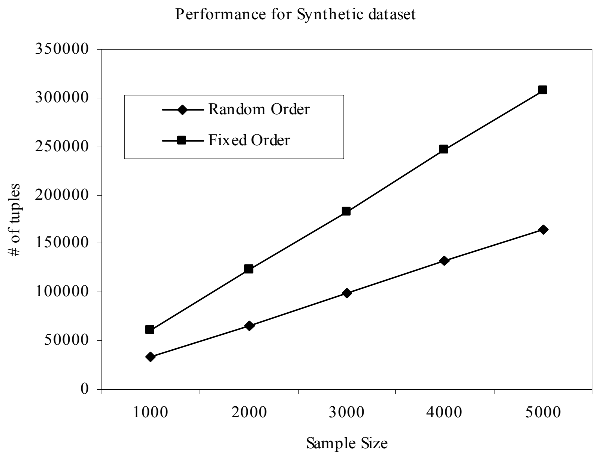

Figure 14: Performance for synthetic dataset with p=0.3

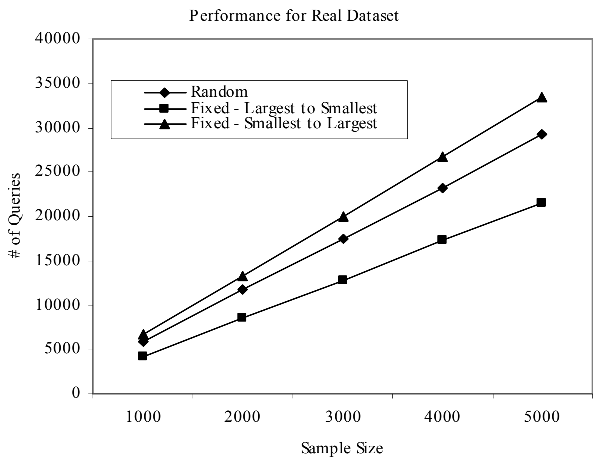

Figure 15: Performance on real large dataset with differing orders

Performance V/S Quality

Figure 16 depicts a comparative view of skew versus number of queries for various values of C. The best performance and quality yield is when C is estimated without any error. As the estimation of C looses its accuracy, performance and quality goes down.

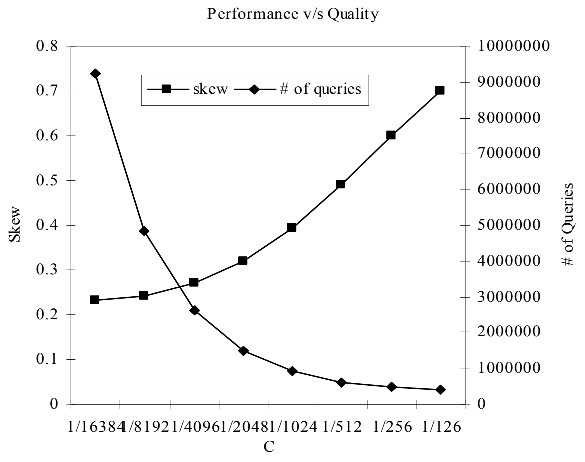

Figure 16: Performance versus Quality measure

HIDDEN-DB-SAMPLER versus Metropolis-Hastings Sampler
A comparative scenario between the HIDDEN-DB-SAMPLER and a competitor suggested by [1] for sampling the index of search engines using Metropolis Hastings technique was evaluated (Figure 17). We implemented this technique on a small mixed Boolean dataset with a top-50 interface. A burn in period of 10 hops was used. The Random Walk based sampler suffered from an inherent skew in that it estimates a rejection sampling phase on the basis of the number of queries that it can form of a single document (our equivalent to a tuple). However, this approach does not work well for the hidden database model since unlike the document model; the space of queries does not differ for individual tuples in the database.

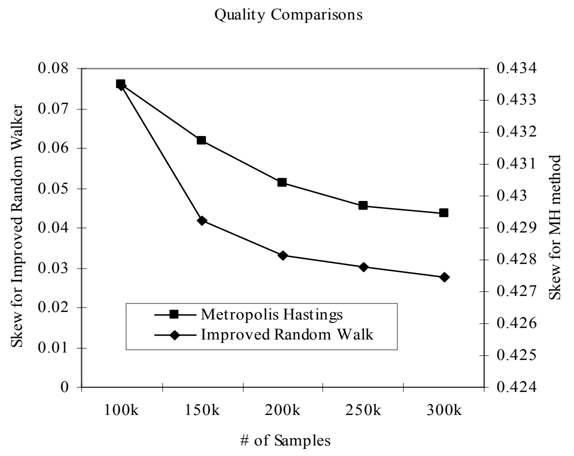

Figure 17: Quality Comparisons

7. CONCLUSION

In this paper we have initiated an investigation of the problem of random sampling of hidden databases on the web. We propose random walk schemes over the query space provided by the interface to sample such databases. We gave simple methods for the sampling and provided some theoretical analysis of the quantitative impact of the ideas on improving efficiency and quality of the resultant samples. We also described a comprehensive set of experiments that demonstrate the effectiveness of our sampling approach.

REFERENCES

[1] Z. Bar-Yossef and M. Gurevich. Random Sampling from a Search Engine's Index. In *Proceedings of WWW*, 2006, 367–376.

[2] K. Bharat and A. Broder. A Technique for Measuring the Relative Size and Overlap of Public Web Search Engines. In *Proceedings of WWW*, 1998, 379–388.

[3] J.P. Callan, M. Connell and A. Du. Automatic Discovery of Language Models for Text Databases. In *Proceedings of ACM SIGMOD*, 1999, 479–490.

[4] S. Chaudhuri, G. Das, and U. Srivastava. Effective Use of Block-Level Sampling in Statistics Estimation. In *Proceedings of ACM SIGMOD*, 2004.

[5] http://www.completeplanet.com/Tutorials/DeepWeb

[6] M. N. Garofalakis and P. B.Gibbons. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of VLDB*, 2001.

[7] W. Hastings. Monte Carlo Sampling Methods using Markov Chains and their Applications. *Biometrika*, 57 (1), 1970, 97–109.

[8] P. J. Haas, C. A. Koenig. Bi-Level Bernoulli Scheme for Database Sampling. In *Proceedings of ACM SIGMOD*, 2004, 275-286.

[9] Ipeirotis, Panagiotis G., L. Gravano, and M. Sahami. Probe, Count, and Classify: Categorizing Hidden Web databases. In *Proceedings of ACM SIGMOD*, 2001.

[10] S. Lawrence and C. L. Giles. Accessibility of Information on the Web. *Nature*, 400, 1999. 107–109.

[11] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280 (5360), 1998.

[12] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. J. *of Chemical Physics*, 21, 1953, 1087–1091.

[13] J. Von Neumann. Various Techniques used in Connection with Random Digits. In *John von Neumann, Collected Works*, Volume V. Oxford, 1963.

[14] F. Olken. Random Sampling from Databases. *PhD Thesis*, University of California, Berkeley, 1993.

[15] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of ACM SIGMOD*, 1984, 256–276.

[16] S. Raghavan and H. Garcia-Molina. "Crawling the Hidden Web," In *Proceedings of VLDB*, 2001, 129-138.

[17] J. Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1), 1985.