Leveraging COUNT Information in Sampling Hidden Databases

Arjun Dasgupta [#], Nan Zhang [*], Gautam Das [#]

[#]*University of Texas at Arlington* , [*]*George Washington University*

{arjundasgupta,gdas} @ uta.edu [#] , nzhang10@gwu.edu [*]

***Abstract*— A large number of online databases are hidden behind form-like interfaces which allow users to execute search queries by specifying selection conditions in the interface. Most of these interfaces return restricted answers (e.g., only top-k of the selected tuples), while many of them also accompany each answer with the COUNT of the selected tuples. In this paper, we propose techniques which leverage the COUNT information to efficiently acquire unbiased samples of the hidden database. We also discuss variants for interfaces which do not provide COUNT information. We conduct extensive experiments to illustrate the efficiency and accuracy of our techniques.**

I. Introduction

A. Hidden Databases

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. These query interfaces allow external users to browse these databases in a controlled manner. Typically users provide inputs in the form interface which are then translated into SQL queries for execution and the results provided to the user on the browser. Databases with such public web-based interfaces are present for many commercial sites, as well as government, scientific, and health agencies.

We focus on two of the simplest and most widely prevalent kind of query interfaces. The first kind of interfaces allows users to specify range conditions on various attributes - however, instead of returning all satisfying tuples, such interfaces restrict the returned results to only a few (e.g., top-k) tuples, sorted by a suitable ranking function. Along with these returned tuples, the interface may also alert the user if there was an "overflow", i.e., if there were other tuples besides the top-k that also satisfied the query conditions but were not returned. We refer to such interfaces as TOP-k-ALERT interfaces. Examples include MSN Stock Screener (*http://moneycentral.msn.com/investor/finder/customstocks.asp*) which has $k = 25$ and Microsoft Solution Finder (*https://solutionfinder.microsoft.com/Solutions/SolutionsDirectory.aspx?mode=search*) which has $k = 500$. The second kind interfaces that we consider are similar to the above, except that instead of simply alerting the user of an overflow, they provide a count of the total number of tuples in the database that satisfy the query condition. We refer to such interfaces as TOP-k-COUNT interfaces. An example is MSN Careers (*http://msn.careerbuilder.com/JobSeeker/Jobs/JobFindAdv.aspx*) which has $k = 4000$.

B. The Problem of Sampling from Hidden Databases

There has been interesting recent focus on the problem of sampling from hidden databases [1]: *given such restricted query interfaces, 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 database and is useful in gathering statistical information about the data. Likewise, random samples from hidden databases can be extremely useful to third-party applications in obtaining insight into the hidden data. However, sampling from hidden databases presents significant challenges as the only view available into these databases is via the proprietary interface that allows only limited access. Thus traditional database sampling techniques that require complete and unrestricted access to the data (e.g., [2], [3]) cannot be easily applied. In [1] an interesting approach named HIDDEN-DB-SAMPLER was proposed for sampling from hidden databases with TOP-k-ALERT interfaces. The approach was based on a random drill-down over the space of all queries executable via the form interface, starting with an extremely broad (therefore overflowing) query, and iteratively narrowing it down by adding random predicates, until a non-overflowing query is reached. Once such a non-overflowing query is reached, one of the returned tuples is randomly picked for inclusion into the sample. This process can be repeated to get samples of any desired size. The paper proposed several variants of HIDDEN-DB-SAMPLER, depending on whether the database consisted of only Boolean attributes or also included categorical attributes. While much of the paper was devoted to sampling from TOP-k-ALERT interfaces, a simple approach for sampling from TOP-k-COUNT interfaces was also proposed.

C. Outline of Technical Results

In this paper we revisit the hidden database sampling problem, and present vastly superior sampling techniques than those proposed in our preliminary work [1]. Unlike our earlier work, our main focus here is on the TOP-k-COUNT interface. However, we also show how a novel hybrid technique can be utilized to also extend our techniques to TOP-k-ALERT interfaces. We briefly describe our new contributions below.

There are two main objectives that any hidden database sampling algorithm should seek to achieve:

- *Sample bias*: Due to the restricted nature of the interface, it is challenging to produce samples that are truly uniform random samples. Consequently, the task is to produce

samples that have small bias, i.e., samples that deviate as little as possible from uniform.
- *Efficiency*: 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 sampling procedure that executes as few queries as possible.

Our Algorithm for TOP-k-COUNT Interfaces: We first discuss our results for TOP-k-COUNT interfaces. As was briefly described in [1], it is fairly straightforward to design a random drill-down procedure that produces samples without any bias. However, the procedure suffers from poor efficiency - it has to execute an inordinate number of queries before obtaining reasonable sized samples. In the current paper we have carefully investigated this problem, and have designed COUNT-DECISION-TREE, a vastly more efficient algorithm. This is one of the principal results of our paper.

The new procedure COUNT-DECISION-TREE is based on two main ideas. The first idea is to continuously log the *query history* while the sampling is in progress - i.e., record (as materialized views) all executed queries and their returned counts. We then design a sampling procedure that tries to leverage the query history as much as possible, where in preparing the next query to execute, preference is given to queries that already appear in the query history, thereby replacing a costly query execution with a fast local look-up at the client's end. In fact, utilization of query history offers opportunities of *query inference* in addition to simple *reuse* - the former refers to queries that may not have been directly executed in the past, but whose counts can be inferred from the ones that have been executed. In the paper we carefully explore our idea of logging query history, and provide both a theoretical analysis of the number of queries saved by this approach, as well as substantial experimental evidence to corroborate our analytical findings.

The second idea is to generalize the notion of attribute ordering used in [1] to that of a *decision tree*. In the earlier work, the random drill-down procedure was guided by an ordering of the attributes, such that each new predicate selected for narrowing the query involved a random value of the next attribute present in the attribute order. For the case of TOP-k-COUNT interfaces, it was suggested that any specific attribute order was adequate for obtaining unbiased samples. In the current paper, we make non-trivial enhancements to this simplistic scheme to obtain unbiased samples, but with significant performance improvements. Our new approach may be considered as the execution of multiple random drill-down procedures (where each such procedure results in the selection of a random sample tuple) except that we always adhere to following paths down a decision tree. In this paper, a decision tree over the database tuples is a tree where all internal nodes are attributes, all leaf nodes are tuples, and each edge leading out of a node is labeled with a unique value from that attribute's domain, along with a transition probability proportional to the number of leaf tuples that can be reached by following that edge. This transition probability is used to select the edge during random drill-down. Each path from the root to a leaf encounters a subset of the attributes in the interface, but the same attribute is never repeated along a path.

Clearly the use of a decision tree is a generalization over using any fixed attribute order - the latter essentially refers to a decision tree that has the same attributes at any given level of the tree. More importantly, while any legitimate decision tree can be used to obtain unbiased samples, the challenge is in designing a decision tree that achieves the maximum efficiency. This problem is complicated by the fact that this tree cannot be created in its entirety, as complete access to all database tuples is impractical; thus this tree has to be built and used on-the-fly, i.e., while the sampling is in progress. Thus if we take a snapshot at any time during the sampling process, we will essentially have created a partial decision tree, with only a few paths extending all the way to the leaves (corresponding to those tuples that have been included in the sample thus far).

Our investigations of such an optimal decision tree-based approach led to several interesting technical results. A theoretical study revealed interesting connections with the seemingly unrelated NP-hard problem of *entity identification* in the design of interactive question-answering systems [4]. Thus we show that drawing samples in an optimal manner using the decision tree approach is computationally intractable. However, our COUNT-DECISION-TREE algorithm is based on a carefully designed heuristic for incrementally building an efficient decision tree while the sampling is in progress, with the goal being that the remaining samples can be obtained in as few queries as possible. This heuristic is based on the online computation of a *savings function* that attempts to select new queries that: (a) leverage the query history and try to reuse as many past queries as possible, and (b) have the best chance of reaching a random tuple as quickly as possible. Although primarily a heuristic, we are able to provide important analytical arguments as to why such a heuristic is expected to do well. Our experiments corroborate our conceptual and analytical arguments to show that COUNT-DECISION-TREE is an order of magnitude more efficient than the previous algorithm presented in [1] for drawing random samples from a TOP-k-COUNT interface.

Our Algorithm for TOP-k-ALERT Interfaces: We next discuss our results for TOP-k-ALERT interfaces. Unlike TOP-k-COUNT interfaces, it is quite difficult to draw a random sample from a TOP-k-ALERT interface without introducing bias into the resultant sample. In fact, bias and efficiency are contradictory goals, and the earlier algorithm HIDDEN-DB-SAMPLER in [1] is actually a parameterized procedure which trades off bias against efficiency. In the current paper we propose a new parameterized procedure, ALERT-HYBRID, for drawing random samples from a TOP-k-ALERT interface which is significantly better than HIDDEN-DB-SAMPLER. This is second main algorithm presented in our paper.

We provide a brief outline of the idea of ALERT-HYBRID. The algorithm consists of two phases. The first phase consists of a drawing a fairly small random sample with very small bias (henceforth called a *pilot sample*) using the earlier HIDDEN-DB-SAMPLER algorithm. Then in the second phase, the remaining desired number of samples is drawn from the alert interface, except that we use our COUNT-DECISION-TREE algorithm to draw the remaining samples. Although

the interface does not have the capability to provide count information for queries, we use the pilot sample to estimate count information for queries. This is done using standard approximate query processing techniques [5]–[7] by executing each query locally on the pilot sample and appropriately scaling the result to estimate the count for the entire database. Interestingly, the "hybrid" idea of using a small amount of pilot samples to bootstrap COUNT-based sampling is inspired by similar sampling approaches considered in other unrelated contexts [8], [9]. Because the counts are only estimates, we are not able to completely remove bias from the resultant sample, however our experiments show that ALERT-HYBRID is significantly better than HIDDEN-DB-SAMPLER for drawing random samples from a TOP-k-ALERT interface - for the same bias (same efficiency), it produces samples more efficiently (with less bias).

In summary, the main contributions of this paper are:

- We revisit the problem of random sampling from hidden databases with proprietary form interfaces.
- We present COUNT-DECISION-TREE, an efficient algorithm for drawing random samples without bias from hidden databases with TOP-k-COUNT interfaces. The algorithm is based on two ideas: (a) the use of query history, and (b) the use of a decision tree. We provide several theoretical insights into the behavior and performance of this algorithm.
- We present ALERT-HYBRID, an efficient algorithm for drawing random samples with small bias from hidden databases with TOP-k-ALERT interfaces. The algorithm is based on using a pilot sample to bootstrap the COUNT-DECISION-TREE algorithm to draw the samples.
- We provide a thorough experimental study that demonstrates the significance of our theoretical results and the superiority of our algorithms over previous efforts.

The rest of this paper is organized as follows. We briefly review the existing sample algorithms for hidden databases in Section 2. In Sections 3 and 4, we introduce our two major algorithms, COUNT-DECISION-TREE and ALERT-HYBRID, respectively. Section 5 presents the experimental results. Related work is reviewed in Section 6, followed by final remarks in Section 7.

II. Preliminaries

A. Models of Hidden Databases

We restrict our discussion in this paper to categorical data - we assume a simple discretization of numerical data to resemble categorical data. Apparently, different discretization will lead to different performance of sampling. How to design an optimal discretization scheme is left as an open problem.

Consider a hidden database table D with m tuples $t_1, \ldots, t_m$ and n attributes $A_1, \ldots, A_n$ with respective domains $Dom_1, \ldots, Dom_n$. The table is only accessible to users through a web interface. We assume a prototypical interface where users can query the database by specifying values for a subset of attributes. Thus a user query Q_S is of the form:

SELECT $*$ FROM D WHERE $A_{i_1} = v_{i_1}$ ... $A_{i_s} = v_{i_s}$, where v_{i_j} is a value from Dom_{i_j}.

Let $Sel(Q_S)$ be the set of tuples in D that satisfy Q_S. As is common with most web interfaces, we shall assume that the query interface is restricted to only return k tuples, where $k \ll m$ is a pre-determined small constant (such as 10 or 50). Thus, $Sel(Q_S)$ will be entirely returned only if $|Sel(Q_S)| \leq k$. If the query is too broad (i.e., $|Sel(Q_S)| > k$), only the top-k tuples in $Sel(Q_S)$ will be selected according to a ranking function, and returned as the query result. The interface will also notify the user that there is an *overflow,* i.e., that not all tuples satisfying Q_S can be returned. At the other extreme, if the query is too specific and returns no tuple, we say that an *underflow* occurs. If there is neither overflow nor underflow, we have a *valid* query result.

For the purpose of this paper, we assume that a restrictive interface does not allow the users to "scroll through" the complete answer $Sel(Q_S)$ when an overflow occurs for Q_S. Instead, the user must pose a new query by reformulating some of the search conditions. We argue that this is a reasonable assumption because many real-world top-k interfaces (e.g., Google) only allow "page turns" for limited (100) times before blocking a user by IP address.

Based on the response provided by the interface when there was an overflow, we classify the interfaces for hidden databases into two categories: TOP-k-ALERT and TOP-k-COUNT. If the interface only issues a Boolean alert i.e., whether there were other tuples besides the top-k that also satisfied the query conditions but were not returned, then the interface is TOP-k-ALERT. If the interface also provides a count of the total number of tuples in the database that satisfy the query condition, we call the interface as TOP-k-COUNT.

B. A Running Example

Table I depicts a simple dataset which we will use as a running example throughout this paper. There are 8 tuples and 7 attributes, including 3 Boolean and 5 categorical with domain size ranging from 4 to 8.

TABLE I
EXAMPLE: INPUT TABLE

	A_1	A_2	A_3	A_4	A_5	A_6	A_7
t_1	0	0	0	0	0	0	0
t_2	0	1	0	0	2	0	1
t_3	1	0	0	1	1	0	2
t_4	1	0	1	1	2	0	3
t_5	2	1	0	0	2	1	4
t_6	2	1	0	1	2	2	5
t_7	3	1	1	1	3	3	6
t_8	4	0	1	1	3	0	7

C. Prior Sampling Algorithms

In this subsection we review three variants of HIDDEN-DB-SAMPLER, the sampling algorithm presented in our earlier work [1] for obtaining random samples from hidden databases.

1. ALERT-ORDER: We first describe a variant that was designed for TOP-k-ALERT interfaces (for the rest of this paper we refer to this variant as ALERT-ORDER). Assume a specific fixed ordering of all attributes, e.g. $A_1, \ldots, A_n$.

Consider Figure 1 a) which represents an *attribute-order tree* over the database tuples, where all internal nodes at the ith level are labeled by attribute A_i. Each internal node A_i has exactly $|Dom_i|$ edges leading out of it, labeled with values from Dom_i. Thus, each path from the root to a leaf represents a specific assignment of values to attributes, with the leaves representing possible database tuples. Note that since some domain values may not lead to actual database tuples, only some of the leaves representing actual database tuples are marked solid, while the remaining leaves are marked empty.

The ALERT-ORDER sampler executes a random walk in this tree to obtain a random sample tuple. To simplify the discussion, assume $k = 1$. Suppose we have reached the ith level and the path thus far represents the query $A_1 = v_1 \& \ldots \& A_{i-1} = v_{i-1}$. The algorithm selects one of the domain values of A_i uniformly at random, say v_i, adds the condition $A_i = v_i$ to the query, and executes it. If the outcome is an underflow (i.e., leads to an empty leaf), 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 random walk may be repeated a number of times to obtain a sample (with replacement) of any desired size. One important point to note is that this method of sampling introduces bias into the sample, as not all tuples are reached with the same probability. Techniques such as *acceptance/rejection sampling* are further employed for reducing bias (see [1] for further details).

For this scheme, clearly the order of the attributes can play an important role in the efficiency of the sampling process. It was suggested in [1] that the attributes be ordered from largest to smallest domain sizes.

2. ALERT-RANDOM: For the special case of Boolean data, since the domain sizes are the same for all attributes, it was suggested that instead of using a specific fixed attribute order, a fresh *random ordering* of attributes be used before every random walk. It was shown that such a scheme helps reduce the bias more than any fixed order attribute scheme. Henceforth we refer to this variant as ALERT-RANDOM.

3. COUNT-ORDER: We now turn our attention to TOP-k-COUNT interfaces. It was pointed out in [1] that, when COUNT information is returned for each query, a random walk scheme can be designed to generate completely unbiased samples. Thus, no bias reduction techniques need to be used later. Henceforth we refer to this variant as COUNT-ORDER. For a given node in the attribute-order tree, instead of choosing edges with uniform probability, COUNT-ORDER chooses an edge with probability proportional to the COUNT of that edge (i.e., proportional to the number of actual tuples that can be reached following that edge). For example, suppose we have reached the ith level and the path thus far represents the query $A_1 = v_1 \& \ldots \& A_{i-1} = v_{i-1}$. Let the current attribute under consideration, A_i, have $|Dom_i| = b_i$ edges labeled with values $v_i^1, \ldots, v_i^{b_i}$. Then, the random walk follows edge v_i^j (i.e., adds

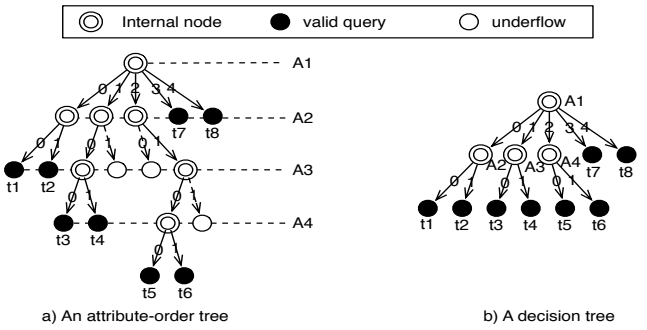

Fig. 1. Attribute-Order Tree vs. Decision Tree

$A_i = v_i^j$ to the query) with probability equal to

$$P(v_i^j) = \frac{\text{COUNT}(A_1 = v_1, \ldots, A_{i-1} = v_{i-1}, A_i = v_i^j)}{\text{COUNT}(A_1 = v_1, \ldots, A_{i-1} = v_{i-1})}.$$

Consider the impact of this approach to the bias of the obtained samples. The probability that a random walk hits a tuple $t = \langle v_1, \ldots, v_n \rangle$ in the database is

$$P(t) = \prod_{i=1}^{n} \Pr\{v_i \text{ is chosen for } A_i\} \quad (1)$$

$$= \prod_{i=1}^{n} \frac{\text{COUNT}(A_1 = v_1, \ldots, A_{i-1} = v_{i-1}, A_i = v_i)}{\text{COUNT}(A_1 = v_1, \ldots, A_{i-1} = v_{i-1})} \quad (2)$$

$$= \frac{1}{m} \quad (3)$$

where, recall that m is the number of tuples in the database and $\text{COUNT}(A_1 = v_1, \ldots, A_{i-1} = v_{i-1}) = \text{COUNT}(*) = m$ for $i = 1$. Thus, the count-based sampling generates unbiased samples.

III. COUNT-DECISION-TREE

In this section we present the main ideas of COUNT-DECISION-TREE, our algorithm for sampling a hidden database with TOP-k-COUNT interface.

A. Motivation

Although the simple COUNT-ORDER algorithm explained in Section II can generate unbiased samples, it also introduces a significant challenge, as the number of queries required for sampling categorical databases may increase dramatically compared with both TOP-k-ALERT algorithms. To understand this, consider a random walk from a node to one of its b successors in the tree. In both TOP-k-ALERT algorithms, an edge is chosen uniformly at random from $[1, b]$, and only one query corresponding to the chosen edge needs to be issued. However in COUNT-ORDER, the counts of *all* edges must be first determined in order to compute their respective transition probabilities, after which an edge is randomly selected to follow. This requires $b - 1$ queries[1]. Thus, COUNT-ORDER

[1]The remaining count can be inferred from these $b-1$ counts and the count of the current node.

may require a large number of queries for sampling categorical databases, especially for attributes with large domains.

The rest of this section is devoted to techniques for improving the efficiency of sampling TOP-k-COUNT interfaces.

We first introduce a generalization of an attribute-order tree to a *decision tree* on the hidden database. The key extension of a decision tree is that it allows each level of the tree to contain different attributes. Figure 1 a) and b) illustrates both types of trees for the database in Table I for the case $k = 1$. Random walks over decision trees are likely to be more efficient than over attribute-order trees, as by leveraging the flexibility of selecting multiple attributes for nodes at the same level, a compact decision tree features a shorter depth and a smaller total number of possible queries. For the example in Figure 1, when one sample tuple needs to be collected, the decision tree provides a saving of $(1/4 \times 1 + 1/4 \times 2) = 3/4$ queries in comparison with the attribute-order tree. We defer a more thorough analysis of the advantages of decision trees over attribute-order trees to Section III-C.

Suppose we are given the *structure* of a decision tree over a hidden database - i.e., the entire tree is available, barring the various COUNT information (or transition probabilities associated with the edges). Figure 2 depicts a count-based sampling algorithm that performs random walks on this decision tree to collect a sample with s tuples (in the figure we use the notation $|u|$ to refer to the count of node (or edge) u, i.e., the number of database tuples below u in the tree). We would like to make several remarks regarding the algorithm. First, this is of course a hypothetical scenario, as such a tree is not available for hidden databases, and in fact has to be constructed on-the-fly (which will be discussed later in the paper). Second, queries corresponding to nodes in the upper level (e.g., root) of the tree may be reused by many random walks, especially if s is large. This motivates us to consider the impact of query history in Section III-B. Third, no matter how the decision tree is structured, the sampling algorithm always generates unbiased samples. Fourth, the number of queries, however, may vary significantly between different structures. An interesting challenge is to identify a structure which achieves the optimal efficiency. We will address this challenge in Sections III-C.

Require: r: root node of the decision tree
1: **for** $i = 1$ to s **do**
2: Obtain the i-th sample as DT_SAMP(r).
3: **end for**
4: **function** DT_SAMP(u)
5: ▷ Let u have b values $v_1, \ldots, v_b$ and edges $u_1, \ldots, u_b$
6: Query $b - 1$ edges for counts of $u_1, \ldots, u_b$.
7: Randomly pick $j \in [1, b]$ s.t. $\Pr\{j \text{ picked}\} = |u_j|/|u|$.
8: **if** $|u_j| \leq k$ **then**
9: **return** a random tuple from the answer to u_j
10: **else**
11: **return** DT_SAMP(u_j).
12: **end if**
13: **end function**

Fig. 2. Sampling TOP-k-COUNT with a Given Decision Tree

B. Improving Efficiency: Query History

We start our discussion on improving the efficiency of count-based sampling by a simple strategy: take advantage of the query history. That is, the sampling algorithm should only send to the hidden database "new" queries which have never been asked before *and* cannot be inferred from the history. An example of inference is the computation of COUNT$(a_1 = 1)$ from COUNT$(*)$ and COUNT$(a_1 = 0)$.

We discuss the impact of leveraging query history to improve sampling efficiency. The following theorem provides a lower bound on the number of queries saved by consulting the query history.

Theorem 3.1: For the algorithm in Figure 2, the number of queries saved by consulting the query history for obtaining s samples ($s \cdot k \ll m$) of a hidden database of size m is at least

$$s_{\text{QH}} > s \cdot \left((b-1) \cdot \log_b s - 2 - \frac{b}{b-1} \right), \qquad (4)$$

where b ($b \geq 2$) is the minimum domain size of an attribute.

We omit the proof due to space limitation. An observation from the theorem is that the saving from history is significant when s is large. For example, obtaining $5,000$ samples from a Boolean database will lead to a saving of at least $41,438$ queries. For a 100,000-tuple i.i.d. Boolean database where the 1's and 0's are uniformly distributed with probability 0.5 each, it implies an expected saving of at least 49.90% when $k = 1$ (from $83,048$ to at most $41,610$). The saving will be even larger for a categorical database with $b > 3$. For example, when $b = 5$, the saving is at least $89,590$ queries. Again, for a 100,000-tuple i.i.d. database with each attribute following uniform distribution on 5 values, it implies an expected saving of 62.62% (from $143,067$ to $53,317$).

Theorem 3.1 provides a lower bound on the number of queries saved by the history. Now consider an even more important problem for leveraging history: *how many unique queries are needed to sample a hidden database with a TOP-k-COUNT interface?* We investigate this problem below. In particular, we consider the maximum number of queries needed in the extreme case where all branches are traversed. The result will also form the foundation for our discussion of building the decision tree in Section III-C.

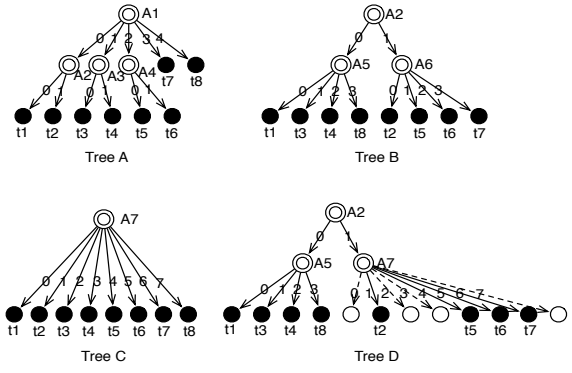

Fig. 3. Examples of Decision Trees

First, we consider a special case of decision trees referred to as *loaded* decision trees. A tree is loaded iff it does not have any empty leaves. For example, of the four trees in Figure 3 corresponding the running example database in Table I, trees A, B, and C are loaded, while tree D is not. In the following, we will first derive the maximum number of unique queries required for sampling a loaded tree. After that, we extend the result to general decision trees.

Theorem 3.2: Given the structure of a loaded decision tree, the total number of unique queries required for obtaining s samples through a TOP-k-COUNT interface is at most $m-1$.

Proof: Let $|L_i|$ and $|\Omega_i|$ be the number of all (internal and leaf) nodes and internal nodes in level i, respectively (root is level 1, let the maximum levels be h). Then, the maximum number of queries issued for level i is $|L_{i+1}|-|\Omega_i|$ because each internal node has one query saved through history inference. Thus, the maximum total number of queries issued is

$$\sum_{i=1}^{h-1}(|L_{i+1}| -| \Omega_i|) = \sum_{i=1}^{h}|L_i| - \sum_{i=1}^{h-1}|\Omega_i| - 1 = m-1. \quad (5)$$

This is due to two reasons. First, $\sum_{i=1}^{h}|L_i| - \sum_{i=1}^{h-1}|\Omega_i|$ is equal to the total number of leaf nodes because all nodes in level h are leaves. Second, the number of leaves is m. ■

The theorem shows that given a loaded decision tree, the maximum number of unique queries required for count-based sampling of the tree only depends on the number of tuples in the database, and *not* by the number of attributes or their domain sizes. For example, trees A, B and C in Figure 3 all have 7 unique queries for count-based sampling: Tree A has 1 at the level 1 and 6 at level 2; Tree B has 4 at level 1 and 3 at level 2; while all 7 queries for Tree C are at the same level.

We now consider the extension to general decision trees. Again, we would like to remark that in practice, we will not be provided with the structure of a decision tree; rather queries must be issued to both construct the decision tree and sample from it. If a decision tree is constructed without consulting the complete database, empty branches often occur and the tree is usually not loaded. Thus, the sampling of a decision tree that has empty leaves is arguably a more practical scenario.

Each edge leading to an empty leaf leads to one additional query, as we can observe from tree D in Figure 3. To analyze the number of empty leaves, we consider an example of m-tuple i.i.d. Boolean dataset studied in [1], where each attribute takes the value of 1 with probability p. Let $L(m,k,p)$ be the expected number of empty leaves for such a dataset. We have

$$L(0,k,p) = 1. \quad (6)$$

$$L(1,k,p) = 0. \quad (7)$$

$$\cdots$$

$$L(k,k,p) = 0. \quad (8)$$

$$L(m,k,p) = \sum_{i=0}^{m}\binom{m}{i}p^i(1-p)^{m-i}(L(i,k,p)+L(m-i,k,p)). \quad (9)$$

Note that although $L(m,k,p)$ appears in both the left and right side of (9), it can nevertheless be solved from the equation. Figure 4 depicts the relationship between the number of empty leaves and the number of tuples when $k=1$. The results are computed from (6)-(9) using Matlab simulation. As we can see, $L(m,p)$ and m roughly follow a linear relationship. Based on (9) and Theorem 3.2, we have the following corollary.

Corollary 3.1: Given an i.i.d. Boolean dataset where each attribute takes the value of 1 with probability p, for all $s \geq 1$, the total number of queries required for obtaining s samples through a top-k interface with COUNT is at most $m-1+L(m,k,p)$.

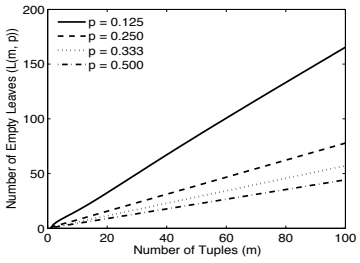

Fig. 4. The number of empty leaves vs. the number of tuples when $k=1$

C. Improving Efficiency: Constructing Decision Tree

1) Motivation and Hardness: Theorem 3.2 indicates that, if a very large number of samples need to be collected, then every decision tree without empty leaves will have the same efficiency because the total number of queries only depends on the size of the database. Nonetheless, the design of the decision tree may play an important role in reality due to the following two reasons:

- Since the number of samples required in practice is usually much smaller than the size of the database, many of the $m-1$ queries may not be issued; thus different decision trees may have different impact on efficiency.
- As we can see from Figure 4, the number of empty leaves may be significant, especially when the attributes skew towards a few values.

We now discuss the design of an efficient decision tree, in particular the following problem: *Given s, the number of samples to be collected, design a decision tree with the minimum sampling cost, i.e., the minimum expected number of queries required to collect s unbiased samples.*

Unfortunately, this problem is hard even if the decision tree can be constructed with full access to the m tuples. Consider a special case of the problem when $s=1$ and $k=1$ for Boolean databases. The problem is essentially the same as computing a decision tree with no empty leaves that has the minimum average path length from root to the leaves. This is equivalent to a well-known problem of constructing an optimal decision tree for the *entity identification* problem [4], for which the following hardness result is known from [4]:

Theorem 3.3: (from Theorem 4.1 in [4]) When $s=1$ and $k=1$, it is NP-hard to construct a decision tree over a Boolean database with the minimum sampling cost, or even approximate it within a factor of $\Omega(\log m)$.

2) Basic Ideas: Due to the hardness of the problem, we propose a heuristic greedy algorithm to construct an efficient decision tree. We remind the reader that the tree cannot be created in its entirety, as complete access to all database tuples is impractical; the tree has to be built and used on-the-fly. At any time during the sampling process, we will essentially have created a partial decision tree, with only a few paths extending all the way to the leaves (corresponding to those tuples that have been included in the sample thus far).

We first discuss the intuition behind the algorithm: the *saving* and *expense* associated with each node in the decision tree. For the ease of understanding, we restrict our attention to $k = 1$ in the discussion of intuition, but will present the algorithm with arbitrary k.

Saving: Recall from the proof of Theorem 3.2 that, when $k = 1$, a decision tree without empty branch requires exactly $m-1$ total queries when the number of queries $s \to \infty$. Consider these as the *baseline queries* for the sampling process. As we mentioned above, the actual number of queries varies from the baseline due to two possible reasons:

- When s is small, a subtree may never be encountered by a random walk. Note that a never-encountered subtree with m tuples yields a reduction of $m-1$ on the number of queries. Let the total reduction be $R(s)$.
- Each empty leaf leads to an increase of 1 on the number of queries. Let the total increase be L.

Thus, the actual number of queries is $m - 1 - (R(s) - L)$. We say that the decision tree yields a *saving* of $R(s) - L$.

Note that unlike the number of baseline queries which is independent of the structure of the decision tree, $R(s) - L$ strongly depends on the tree structure. For example, consider trees in Figure 3. Tree C offers no saving at all, because all possible queries will be issued to collect the first sample. When $s = 1$, the saving of tree B is 3 because one of the 2nd level nodes (A5 and A6) cannot be encountered by the random walk.

The saving also depends on s. When s increases, the saving of tree B decreases rapidly because it is very likely that both A5 and A6 will be encountered. Nonetheless, tree A might still offer some saving if one of the three nodes (A2, A3, A4) are not encountered by random walks. Thus, a critical challenge is to construct a decision tree with maximized saving given s.

Consider the saving associated with not reaching a node u of A_i (but reaching its ancestors). Denote such saving by $R(s,u) - L(u)$. Consider $R(s,u)$ first. Recall that b_i is the domain size of a_i. Let u_j be the edge of u which corresponds to the j-th value of A_i, and $|u_j|$ be the number of tuples below u_j. Define

$$R(s,u) = \sum_{j=1}^{b_i} \Pr\{u_j \text{ is not traversed}, u \text{ is reached}\} \cdot (|u_j| - 1)$$
$$= \sum_{j=1}^{b_i} \left(\left(1 - \frac{|u_j|}{m}\right)^s - \left(1 - \frac{|u|}{m}\right)^s \right) \cdot (|u_j| - 1),$$

and

$$L(u) = |\{j | j \in [1, b_i], |u_j| = 0\}|. \tag{10}$$

It is easy to see that $R(s) = \sum_u R(s,u)$, $L = \sum_u L(u)$, and

$$R(s) - L = \sum_u (R(s,u) - L(u)). \tag{11}$$

We refer to $R(s,u) - L(u)$ as the *saving* of u. Table II shows the saving of the root node for trees A, B and C in Figure 3. As we can see, tree B offers the greatest saving when $s = 1$, but its saving decreases rapidly to below tree A when s increases to > 3. As we discussed above, tree C has a saving of 0.

TABLE II
EXAMPLE: SAVING

s	1	2	3	4	5	6
Tree A	2.2500	1.6875	1.2656	0.9492	0.7119	0.5339
Tree B	6.0000	3.0000	1.5000	0.7500	0.3750	0.1875
Tree C	0	0	0	0	0	0

Expense: The saving function $R(s) - L$ concerns how many queries are saved from the baseline $m - 1$ queries. Now consider the opposite view: how many queries are executed, starting from 0 queries? Let C be this number. Note that $C = m - 1 - (R(s) - L)$. Define

$$C(u) = \Pr\{u \text{ is reached}\} \cdot (b_i - 1) \tag{12}$$
$$= \left(1 - \left(1 - \frac{|u|}{m}\right)^s\right) \cdot (b_i - 1). \tag{13}$$

Again, we have $C = \sum_u C(u)$. We refer to $C(u)$ as the *expense* of u.

Intuition of Constructing a Decision Tree: The task of constructing a decision tree is essentially to select an attribute label for each node: first, select an attribute for the root, and then recursively choose an attribute for each child, and so on. During the process, we aim to increase $\sum_u (R(s,u) - L(u))$ and reduce $\sum_u C(u)$. However, note that the total number of nodes depends on the structure of the decision tree, and may not be known during the construction. Thus, while selecting an attribute for a node u, we propose a heuristic of maximizing the *saving per expense* ratio

$$SER(s,u) = \frac{R(s,u) - L(u)}{C(u)}. \tag{14}$$

Due to the constraint that $\sum_u (R(s,u) - L(u) + C(u)) = m - 1$, limiting the ratio also limits the number of queries issued for sampling. In particular, we have the following theorem:

Theorem 3.4: If all nodes in the decision tree satisfies $SER(s,u) \geq \sigma$, then the expected number of queries for obtaining s samples is at most $(m-1)/(\sigma+1)$.

For data in Table I, Table III shows the SER of different attributes for choosing the root node when $s = 1$ and $s = 10$. Note that A_7 is not shown in the table because its SER is always 0. As we can see, A_2 will be chosen as the root when $s = 1$, while A_1 will be chosen when $s = 10$. This is consistent with our intuition discussed above.

Computation of $SER(s,u)$: For computing $SER(s,u)$, four variables are needed: the number of samples s, the COUNT of the current node $|u|$, the domain size b_i and the branch counts $|u_j|$. Among them, $|u|$ and s are already determined, while b_i

TABLE III
EXAMPLE: SER FOR THE ROOT NODE

	A_1	A_2	A_3	A_4	A_5	A_6
$s = 1$	0.5625	3.0000	2.7500	2.7500	0.7500	0.5000
$s = 10$	0.0422	0.0059	0.0184	0.0184	0.0197	0.0001

and $|u_j|$ depend on the selected attribute. b_i can be learned through domain knowledge. However, $|u_j|$ have to be queried from the hidden database. For high-domain-size attributes, $|u_j|$ requires a large number of queries, which jeopardize our ultimate objective of minimizing the total number of queries.

Fortunately, the exact computation of $SER(s, u)$ might not be necessary for our algorithm. Note that to select an attribute for node u, we only need to determine which attribute returns the largest $SER(s, u)$. An important observation is that $R(s, u) - L(u)$ may vary significantly between attributes of different domain size. For example, consider the selection between two attributes A_1, A_2 for the root node. Both follow uniform distribution with domain size $b_1 = 2$ and $b_2 = 10$. Note that when $m \gg 10$, neither of them is likely to have $L(u) > 0$. Thus,

$$SER(s, a_1) = \frac{m-2}{2^s} \ll \frac{(m-10) \cdot 9^s}{9 \cdot 10^s} = SER(s, a_2).$$

Clearly, in this case, a rough estimation of $|u_j|$ would be sufficient for choosing between the two attributes.

We leverage this property of $SER(s, A_i)$ by approximating its value with the minimum number of queries. The simplest choice is to assume that all attributes follow the uniform distribution, and to compute $|u_j| = |u|/b_i$. However, we found through experiments that this approximation is oversimplified because many attributes in real-world datasets have highly skewed value distribution.

Thus, we propose to first issue a small number ($\sum_i b_i$) of *marginal* queries, and then estimate $|u_j|$ based on the conditional independence assumption: The marginal queries are COUNT($A_i = v_1$), ..., COUNT($A_i = v_{b_i}$) for all attributes A_i. To select an attribute for node u, we estimate the COUNT of branch u_j for attribute A_i by

$$|u_j|_e = |u| \cdot \frac{COUNT(a_i = v_j)}{COUNT(*)}. \tag{15}$$

However, note that once an attribute a_i is selected, we will actually query all $|u_j|$ in order to determine the probability for following each branch. By doing so, we save the queries used for constructing but not sampling the decision tree (i.e., queries $|u_j|$ for attributes which are not eventually chosen), without affecting the unbiasedness of the collected samples.

D. Algorithm COUNT-DECISION-TREE

Figure 5 depicts COUNT-DECISION-TREE, our algorithm for sampling TOP-k-COUNT interfaces. It performs the following alternative steps: a) determine the attribute for the current node (Lines 1 to 3), then b) determine which branch to follow, and so on. The estimation of $SER(s, u)$ is used to determine the attribute (Line 6). Note that once an attribute is chosen for a node, it is available for reuse for future samples in order to leverage the query history. Determining the next edge involves the execution of $b_i - 1$ queries (Line 8), followed by a random picking of the next edge (Line 10).

Require: Attr($\cdot$) = $\emptyset$ if not assigned
1: **for** $i = 1$ to s **do**
2: Obtain the i-th sample as DT_SAMP($s - i + 1$, $\langle\rangle$).
3: **end for**
4: **function** DT_SAMP(s_t, $path$)
5: **if** Attr($path$) = ϕ **then**
6: Attr($path$) = $\arg\max((R(s_t, u, k) - L(u, k))/C(u))$.
7: **end if**
8: Query $b-1$ branches. (Only issue those not in history)
9: Randomly pick $j \in [1, b]$ s.t. $\Pr\{j \text{ picked}\} = |u_j|/|u|$.
10: **if** $|u_j| \leq k$ **then**
11: **return** a random tuple from the answer to u_j
12: **else**
13: **return** DT_SAMP(s_t, $path \| \text{Attr}(path) = v_j$).
14: **end if**
15: **end function**

Fig. 5. COUNT-DECISION-TREE

COUNT-DECISION-TREE also extends the previous discussion by addressing the cases with interface parameter $k > 1$. Clearly, the value of $C(u)$ is unaffected. For computing the saving $R(s, u) - L(u)$, we define the number of baseline queries as $m/k - 1$. Thus, the saving becomes

$$R(s, u, k) = \sum_{j=1}^{b_i} \left(\left(1 - \frac{|u_j|}{m}\right)^s - \left(1 - \frac{|u|}{m}\right)^s \right) \cdot \left(\frac{|u_j|}{k} - 1 \right),$$

and

$$L(u, k) = \sum_{j | j \in [1, b_i], |u_j| < k} \frac{k - u_j}{k}. \tag{16}$$

IV. ALERT-HYBRID

In this section, we present the main ideas behind ALERT-HYBRID, our new algorithm for sampling hidden database behind a TOP-k-ALERT interface.

A. Basic Ideas

A major problem of ALERT-ORDER, the state-of-the-art algorithm for sampling TOP-k-ALERT interfaces, is the bias of the collected samples. Since ALERT-ORDER chooses each branch of a node with equal probability, those tuples on upper levels of the tree (which require shorter walk from the root) are more likely to be sampled. Although an acceptance-rejection module was introduced to reduce the bias [1], not many samples can be rejected in order to maintain the efficiency of ALERT-ORDER. As a result, the remaining bias may still be significant, as we will illustrate in the experiments.

On the other hand, the algorithms we just discussed for TOP-k-COUNT interfaces generate no bias because each branch is chosen with probability proportional to its COUNT. As a result, each tuple is sampled with equal probability.

Clearly, the COUNT information which is absent from TOP-k-ALERT interfaces can play an important role on reducing the bias of collected samples.

Fortunately, the COUNT information is not completely out of reach in TOP-k-ALERT interfaces. In particular, after a small number of samples are collected, the COUNT of certain queries may be *estimated* from the collected samples.

Thus, we propose ALERT-HYBRID, a two-phase procedure by which the sampler first collects a small number (say s_1) of *pilot samples* for COUNT estimations, and then use the estimated COUNT to facilitate the collection of the remaining (much larger) $s - s_1$ samples. The s_1 samples can be simply collected by ALERT-ORDER, parameterized to produce samples with small bias. The small bias in the s_1 samples is desirable because it helps in accurate COUNT estimations in the second phase. Although this requirement makes the ALERT-ORDER procedure less efficient, the relatively small number of the pilot samples required ensures that the cost of the first phase is a modest portion of the overall sampling cost.

For the remaining $s - s_1$ samples, note that we cannot directly use the COUNT-DECISION-TREE algorithm because not all nodes can have COUNT accurately estimated from a very small number ($s_1 \ll m$) of samples. Thus, we propose a *hybrid* approach which integrates COUNT-based and ALERT-based sampling. In particular, after collecting the s_1 samples, we invoke the COUNT-DECISION-TREE algorithm until reaching a node u with COUNT in the collected samples less than a threshold, say $c_{\rm S}$. At this node, there are not enough collected samples to support a robust estimation of the probability for following each edge. Thus, a natural choice is to switch to ALERT-based sampling. In particular, ALERT-ORDER is called to collect a sample under node u. As we can see, this hybrid approach starts with COUNT-based sampling at the upper levels of the tree, and then switches to ALERT-based when there is not enough support from the collected samples. Initially, the switch from COUNT-based to ALERT-based sampling may occur early at the upper levels. However, when more samples are collected (at the second phase), more nodes will be able to support COUNT-based sampling, and thus the switch may occur later.

There are two important parameters in the algorithm: s_1, the number of pilot samples collected for initial count estimation, and $c_{\rm S}$, the count threshold for switching to ALERT-ORDER. The setting of s_1 influences the efficiency of ALERT-HYBRID for two reasons: First, with a small s_1, the constructed decision tree is unlikely to be optimal, and therefore may require more queries in the second phase. Second, if s_1 is too large, there will be a large number of queries spent in collecting the pilot samples. Note that these queries are unlikely to be reused in the second phase because COUNT-DECISION-TREE may use a different tree from the attribute-order tree used by ALERT-ORDER in the first phase.

The setting of $c_{\rm S}$ influences the bias of the collected samples. Note that in the count-based sampling part of the tree, the probability of following each branch is determined by the COUNT information estimated from the samples. Thus, error on the estimated COUNT will lead to biased samples. Thus, the value of $c_{\rm S}$ should be large enough to enable a stable estimation for the probability of following each edge. Nonetheless, if $c_{\rm S}$ is too large, a random walk might switch to ALERT-ORDER at very early stage of a random walk, and thereby introduce more bias to the samples.

We will discuss the impact of different settings of s_1 and $c_{\rm S}$ in greater details in the experimental results section. Nonetheless, we would to remark that, although the experimental results verify the effect of s_1 and $c_{\rm S}$ on the efficiency and bias of ALERT-HYBRID, for the class of datasets we tested, the efficiency and bias are not very sensitive to s_1 and $c_{\rm S}$ as long as the parameters are set within a reasonable range. How to determine the optimal values for s_1 and $c_{\rm S}$ is left as an open problem for future work.

B. Algorithm ALERT-HYBRID

Figure 6 depicts the detailed algorithm for ALERT-HYBRID. In the algorithm, $T_{\rm S}$ is the set of collected samples (to which a newly acquired sample is appended); and $T_{\rm S}(path)$ (resp. $T(path)$) is the subset of tuples in $T_{\rm S}$ (resp. T) which satisfy the selection conditions in $path$.

The basic steps can be stated as follows. First, the algorithm collects s_1 pilot samples before using the hybrid sampling method to collect the other samples. During the hybrid sampling, ALERT-ORDER is used when the current node has COUNT less than $c_{\rm S}$ in $T_{\rm S}$. Otherwise, COUNT-DECISION-TREE is used, with the only difference that the counts of the current node (i.e., $|u|_e$) and all edges (i.e., $|u_j|_e$) are estimated from the samples rather than queried from the database. Clearly, the saving function $R_e(s_{\rm t}, u) - L_e(u) - C_e(u)$ is estimated as well. Both the pilot samples and the samples collected by hybrid sampling are returned.

```
1: for i = 1 to s_1 do
2:     T_S[i] ← ALERT-ORDER(T).
3: end for
4: for i = s_1 to s do
5:     T_S[i] ← HYBRID_SAMP(s − i + 1, ⟨⟩).
6: end for
7: function HYBRID_SAMP(s_t, path)
8:     if COUNT(T_S(path)) < c_S then
9:         return ALERT-ORDER(T(path))
10:    else if Attr(path) = φ then
11:        Attr(path) = arg max R_e(s_t, u) − L_e(u) − C_e(u).
12:    end if
13:    Randomly pick j ∈ [1, b] with P(j) = |u_j|_e/|u|_e.
14:    Query q = (path‖Attr(path) = v_j).
15:    if q is a valid query then
16:        return a random tuple from the answer to q
17:    else
18:        return HYBRID_SAMP(s_t, path‖Attr(path) = v_j).
19:    end if
20: end function
```

Fig. 6. ALERT-HYBRID

V. Experimental Results

In this section, we describe our experimental setup, compare our two algorithms with the existing ALERT-RANDOM,

ALERT-ORDER, and COUNT-ORDER algorithms, and draw conclusions on the impact of our three main ideas: leveraging query history, constructing an efficient decision tree, and sampling TOP-k-ALERT interfaces with ALERT-HYBRID. Note that the existing algorithms for comparison were proposed as the HIDDEN-DB-SAMPLER in [1].

A. Experimental Setup

1) Hardware: All experiments were on a machine with Intel Xeon 2GHz CPU with 4GB RAM and Windows XP operating system. All our algorithms were implemented using C# and Matlab.

2) Datasets: We conducted the experiments on three types of datasets: *Boolean Synthetic*, *Yahoo! Auto*, and *Census*. For all datasets, we tested a TOP-k-COUNT interface with $k = 10$.

Boolean Synthetic: Two Boolean synthetic datasets were generated. Both have $200,000$ tuples. The first one is generated as i.i.d. data having 80 attributes with the probability of 1 being 25%. We refer to this dataset as the *Boolean-i.i.d.* dataset. The second dataset is generated in a way such that different attributes have diverse distribution. In particular, there are 40 independent attributes, 5 of which have uniform distribution, while the others have the probability of 1 ranging from $1/160$ to $35/160$ with step of $1/160$. We refer to this dataset as the *Boolean-mixed* dataset.

Yahoo! Auto: The Yahoo! Auto (YA) dataset consists of data crawled from a real-world hidden database at *http://autos.yahoo.com/*. In particular, it contains 15,211 used cars for sale in the Dallas-Fort Worth metropolitan area. There are 32 Boolean attributes, such as A/C, Power Locks, etc, and 6 categorical attributes, such as Make, Model, Color, etc. The domain size of categorical attributes ranges from 5 to 447.

Census: The Census dataset consists of the 1990 US Census Adult data published on the UCI Data Mining archive. After removing attributes with domain size greater than 100, the dataset had 12 attributes and 32,561 tuples. It is instructive to note that the domain size of the attributes of the underlying data is unbalanced in nature. The attribute with the highest domain size has 92 categories and the lowest-domain-size attributes are Boolean.

3) Parameter Settings: The experiments involve five algorithms. Among them, ALERT-RANDOM and COUNT-DECISION-TREE are parameter-less. ALERT-ORDER requires a parameter called scaling factor C for the acceptance/rejection module, in order to tradeoff between efficiency and bias. Following the heuristic in [1], for Boolean datasets, we set $C = 1/2^l$ where l is the average length of random walks for collecting the samples. For categorical data, we consider various values of C to tradeoff between efficiency and bias. COUNT-ORDER requires input of an (arbitrary) attribute order. We randomly generate the order in our experiments. Our ALERT-HYBRID approach requires two parameters: the number of pilot samples s_1 and the switching count threshold c_{S}. We set $s_1 = 100$ and $c_{\mathrm{S}} = 10$ by default, but conducted experiments with various other combinations.

4) Performance Measures: For each algorithm, there are two performance measures: *efficiency* and *bias*. Efficiency of a sampling algorithm was measured by counting the number of queries that were executed to reach a certain desired sample size. To measure the bias of collected samples, we use the same measure as [1] which compares the marginal frequencies of attribute values in the original dataset and in the sample:

$$bias = \sqrt{\frac{\sum_{v \in V} \left(1 - \frac{p_S(v)}{p_D(v)}\right)^2}{|V|}}. \tag{17}$$

Here V is a set of values with each attribute contributing one 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 unbiased uniform random sample, then the relative frequency of any value will be the same as in the original dataset. However, note that even for uniform random samples, this method of measuring bias will result in small but possibly non-zero bias.

B. Comparison with Existing Algorithms

1) COUNT-DECISION-TREE: We compared the performance of COUNT-DECISION-TREE with three existing algorithms: COUNT-ORDER, ALERT-ORDER, and ALERT-RANDOM (note: although the latter two algorithms are designed for ALERT interfaces, they can sample from COUNT interfaces by ignoring the returned counts).

For COUNT-ORDER, our direct competitor for COUNT interfaces, we conducted the comparison on both Yahoo! Auto and Census datasets. The number of queries issued are shown in Figures 7. Note that both algorithms generate unbiased samples. As we can see, our algorithm requires orders of magnitude fewer queries than COUNT-ORDER.

For ALERT-ORDER, we conducted the comparison on the categorical Census dataset. In particular, we tested ALERT-ORDER with two settings of the scaling factor [1]: $C = 1/15000$ and $C = 1/400000$. The number of queries issued and the bias of samples collected are shown in Figures 8 and 9, respectively. As we can see, our algorithm significantly outperforms both settings of ALERT-ORDER in efficiency and bias (recall that even though our measurements show non-zero marginal bias, technically COUNT-DECISION-TREE has no bias).

Since ALERT-RANDOM was designed for Boolean datasets [1], we performed the comparison on the Boolean-mixed dataset. As seen in Figures 10 and 11, our algorithm significantly outperforms ALERT-RANDOM in both efficiency and bias.

2) ALERT-HYBRID: We compared the performance of ALERT-HYBRID with both existing algorithms for ALERT interfaces: ALERT-RANDOM and ALERT-ORDER. Figures 12 and 13 shows results on the Boolean-i.i.d. dataset. We see that ALERT-HYBRID requires significantly fewer queries than both of the previous approaches, and produces substantially less bias than ALERT-ORDER.

C. Effects of History and Decision Tree for COUNT-DECISION-TREE

The above subsection illustrates the improvement of our COUNT-DECISION-TREE algorithm over the prior algorithms. The improvement comes from a combination of two

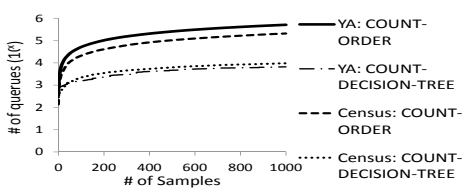

Fig. 7. Number of queries vs. samples for COUNT-DECISION-TREE and COUNT-ORDER

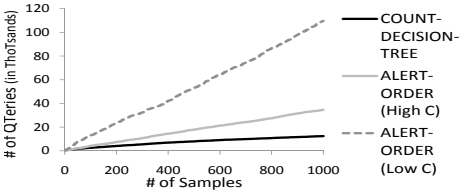

Fig. 8. Number of queries vs. samples for COUNT-DECISION-TREE and ALERT-ORDER

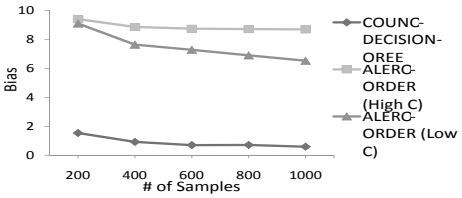

Fig. 9. Bias vs. Number of samples for COUNT-DECISION-TREE and ALERT-ORDER

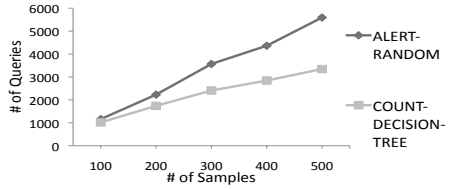

Fig. 10. Number of queries vs. samples for COUNT-DECISION-TREE and ALERT-RANDOM

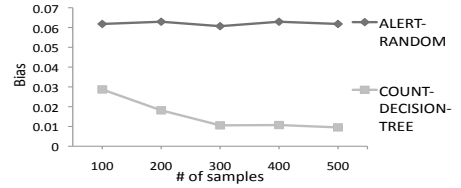

Fig. 11. Bias vs. Number of samples for COUNT-DECISION-TREE and ALERT-RANDOM

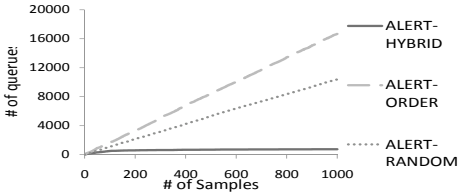

Fig. 12. Number of queries vs. Number of samples for ALERT-RANDOM, ALERT-ORDER, and ALERT-HYBRID

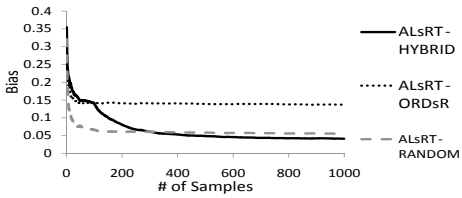

Fig. 13. Bias vs. Number of samples for ALERT-RANDOM, ALERT-ORDER, and ALERT-HYBRID

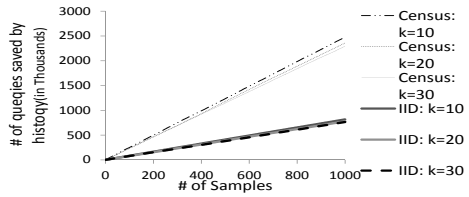

Fig. 14. Number of queries saved by history vs. Number of samples for COUNT-DECISION-TREE

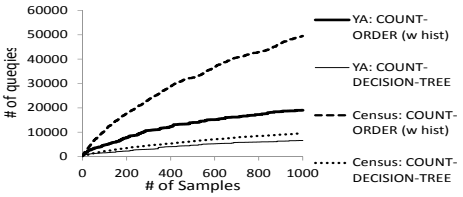

Fig. 15. Number of queries vs. Number of samples for COUNT-DECISION-TREE and COUNT-ORDER with history.

techniques: query history and decision tree. In this subsection, we illustrate the effect of each technique separately.

First, we consider the effect of query history on the performance of COUNT-DECISION-TREE. We conducted the experiments on both categorical (Census) and Boolean (i.i.d.) datasets. Figure 14 depicts the number of queries saved by considering history. As we can see, the saving is roughly linear to the number of samples, and is not sensitive to the value of k. This is consistent with our intuition from Theorem 3.1.

Then, we consider the effect of decision tree construction on the performance of COUNT-DECISION-TREE. To remove the effect of history, we added the technique of history saving to COUNT-ORDER, and then compared its efficiency with COUNT-DECISION-TREE. The results for Yahoo! Auto and Census datasets are shown in Figures 15. As we can see, for collecting 1,000 samples, we achieve 289% and 520% improvement on efficiency for Yahoo! Auto and Census datasets, respectively, while providing unbiased samples.

D. Analysis of ALERT-HYBRID

We first consider the effect of using pilot samples to bootstrap COUNT-based sampling in ALERT-HYBRID. In particular, we compare its efficiency with ALERT-ORDER *after* adding the technique of history saving to ALERT-ORDER. The result for the Boolean-i.i.d. dataset is shown in Figure 16. As we can see, the "hybrid" technique by itself not only reduces bias (as shown in Figure 13), but also significantly improves the sampling efficiency (by 188% when $s = 1,000$).

An interesting observation from Figure 16 is that most queries issued by ALERT-HYBRID are for collecting the pilot samples. After the pilot samples are collected, the number of queries per remaining sample is much lower than that of the ALERT-ORDER and ALERT-RANDOM algorithms. The reason is that no query needs to be issued for a node that enables COUNT-based sampling. Clearly, we can expect the efficiency improvement of ALERT-HYBRID to be even more significant as the number of samples becomes larger.

We now consider the effect of the two parameters s_1 and c_S on the performance of ALERT-HYBRID. Figure 17 shows the change of efficiency and bias when s_1 ranges from 50 to 250 and c_S is fixed at 10. As we can see, increase on s_1 reduces bias, because the larger number of pilot samples delays the switching to ALERT-ORDER which generates higher bias. On the other hand, the greater s_1 is, the more queries need to be issued because the queries used to obtain the pilot samples are unlikely to be reused during hybrid sampling.

Figure 18 shows the change of efficiency and bias when c_S ranges from 1 to 25 and s_1 is fixed at 100. As we can see, when c_S is too low (e.g., 1), the bias is high because the estimated count used for count-based sampling has a high error. Nonetheless, when c_S is too large, the bias becomes higher again because switching to ALERT-ORDER in the hybrid sampling phase occurs earlier which introduces higher bias. This is consistent with our discussion in Section IV.

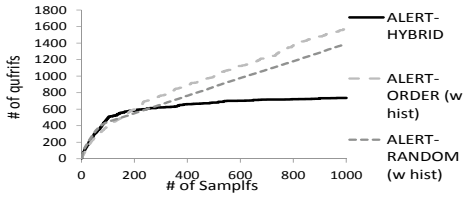

Fig. 16. Number of queries vs. Number of samples for ALERT-HYBRID, ALERT-RANDOM, and ALERT-ORDER with history.

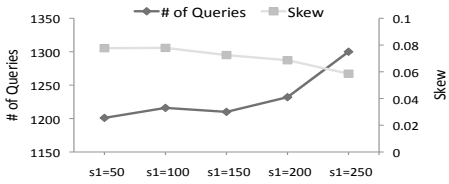

Fig. 17. Number of queries and bias vs. s_1 for ALERT-HYBRID

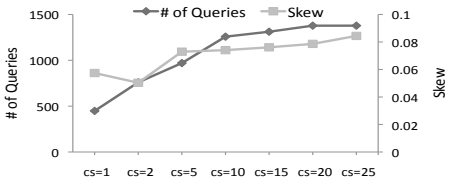

Fig. 18. Number of queries and bias vs. c_S for ALERT-HYBRID

VI. Related Work

Crawling and Sampling from Hidden Databases: There has been prior work on crawling as well as sampling hidden databases using their public search interfaces. Several papers have dealt with the problem of crawling and downloading information present in hidden text based databases [10]–[12]. [13]–[15] deal with extracting data from structured hidden databases. [16] and [17] use query based sampling methods to generate content summaries with relative and absolute frequencies while [18], [19] uses two phase sampling method on text based interfaces. On a related front [20], [21] discuss top-k processing which considers sampling or distribution estimation over hidden sources. A closely related area of sampling from a search engines index using a public interface has been addressed in [12] and more recently [22], [23]. In [1] the authors have developed techniques for random sampling from structured hidden databases leading to the HIDDEN-DB-SAMPLER algorithm. The hybrid technique used in ALERT-HYBRID has also been used for other sampling applications such as block-level sampling [8] and sampling peer-to-peer networks [9].

Approximate Query Processing and Database Sampling: Approximate query processing (AQP) for decision support, especially sampling-based approaches for relational databases, has been the subject of extensive recent research; e.g., see tutorials by Das [5] and Garofalakis et al [6], as well as the report [7] and the references therein.

VII. Conclusion

In this paper, we investigated techniques which leverage the COUNT information to efficiently acquire unbiased samples of hidden databases. In particular, we proposed the COUNT-DECISION-TREE algorithm based on two ideas: (a) the use of query history, and (b) the construction and use of an efficient decision tree. We also discuss variants for TOP-k-ALERT interfaces which do not provide COUNT information. In particular, we presented ALERT-HYBRID based on using pilot samples to bootstrap the COUNT-DECISION-TREE algorithm draw the samples. Our thorough experimental study demonstrates the superiority of our sampling algorithms over the existing algorithms.

Acknowledgment

We would like to thank the anonymous reviewers for useful comments. This work was supported in part by the US National Science Foundation under Grants 0845644, 0852674, and 0852673, unrestricted gifts from Microsoft Research, and start-up funds from the University of Texas at Arlington. Any opinions, findings, conclusions, and/or recommendations in this material, either expressed or implied, are those of the authors and do not necessarily reflect the views of the sponsors listed above.

References

[1] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.
[2] F. Olken and D. Rotem, "Random sampling from databases - a survey," *Statistics & Computing*, vol. 5, no. 1, pp. 25–42, 1995.
[3] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.
[4] V. T. Chakaravarthy, V. Pandit, S. Roy, P. Awasthi, and M. Mohania, "Decision trees for entity identification: approximation algorithms and hardness results," in *PODS*, 2007, pp. 53–62.
[5] G. Das, "Survey of approximate query processing techniques (tutorial)," in *SSDBM*, 2003.
[6] M. N. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes," in *VLDB*, 2001.
[7] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik, "The new jersey data reduction report," *IEEE Data Engineering Bulletin*, vol. 20, no. 4, pp. 3–45, 1997.
[8] S. Chaudhuri, G. Das, and U. Srivastava, "Effective use of block-level sampling in statistics estimation," in *SIGMOD*, 2004.
[9] B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki, "Efficient approximate query processing in peer-to-peer networks," *TKDE*, vol. 19, no. 7, pp. 919–933, 2007.
[10] E. Agichtein, P. G. Ipeirotis, and L. Gravano, "Modeling query-based access to text databases," in *WebDB*, 2003.
[11] A. Ntoulas, P. Zerfos, and J. Cho, "Downloading textual hidden web content through keyword queries," in *JCDL*, 2005.
[12] K. Bharat and A. Broder, "A technique for measuring the relative size and overlap of public web search engines," in *WWW*, 1998.
[13] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB*, 2001.
[14] S. W. Liddle, D. W. Embley, D. T. Scott, and S. H. Yau, "Extracting data behind web forms," in *ER (Workshops)*, 2002.
[15] M. Alvarez, J. Raposo, A. Pan, F. Cacheda, F. Bellas, and V. Carneiro, "Crawling the content hidden behind web forms," in *ICCSA*, 2007.
[16] J. P. Callan and M. E. Connell, "Query-based sampling of text databases," *ACM Transactions on Information Systems*, vol. 19, no. 2, pp. 97–130, 2001.
[17] L. G. Panagiotis G. Ipeirotis, "Distributed search over the hidden web: Hierarchical database sampling and selection," in *VLDB*, 2002.
[18] Y.-L. Hedley, M. Younas, A. E. James, and M. Sanderson, "A two-phase sampling technique for information extraction from hidden web databases," in *WIDM*, 2004.
[19] ——, "Sampling, information extraction and summarisation of hidden web databases," *Data and Knowledge Engineering*, vol. 59, no. 2, pp. 213–230, 2006.
[20] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD*, 2002.
[21] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases," in *ICDE*, 2002.
[22] L. Barbosa and J. Freire, "Siphoning hidden-web data through keyword-based interfaces," in *SBBD*, 2004.
[23] Z. Bar-Yossef and M. Gurevich, "Random sampling from a search engine's index," in *WWW*, 2006.