Turbo-Charging Hidden Database Samplers with Overflowing Queries and Skew Reduction

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

Nan Zhang [*]
George Washington University
nzhang10@gwu.edu

Gautam Das [†]
University of Texas at Arlington
gdas@uta.edu

ABSTRACT

Recently, there has been growing interest in random sampling from online hidden databases. These databases reside behind form-like web interfaces which allow users to execute search queries by specifying the desired values for certain attributes, and the system responds by returning a few (e.g., top-k) tuples that satisfy the selection conditions, sorted by a suitable scoring function. In this paper, we consider the problem of uniform random sampling over such hidden databases. A key challenge is to eliminate the skew of samples incurred by the selective return of highly ranked tuples. To address this challenge, all state-of-the-art samplers share a common approach: they do not use overflowing queries. This is done in order to avoid favoring highly ranked tuples and thus incurring high skew in the retrieved samples. However, not considering overflowing queries substantially impacts sampling efficiency.

In this paper, we propose novel sampling techniques which do leverage overflowing queries. As a result, we are able to significantly improve sampling efficiency over the state-of-the-art samplers, while at the same time substantially reduce the skew of generated samples. We conduct extensive experiments over synthetic and real-world databases to illustrate the superiority of our techniques over the existing ones.

1. INTRODUCTION

In this paper we consider the problem of random sampling over online hidden databases. A hidden database, such as Google Base [16] and Yahoo! Auto [22], allows external users to access its contents via a restricted form-like web interface. This restricted query interface primarily allows users to execute search queries by selecting the desired values for one or more attributes, e.g.,

`SELECT * FROM D WHERE` $a_1 = v_1$ `AND` $a_3 = v_3$

and the system responds by returning a few (e.g., top-k where k is a small constant such as 20 or 50) tuples that satisfy the selection conditions, sorted by a suitable scoring function. Along with these returned tuples, the interface also usually alerts the user if there was an "overflow", i.e., if there are other tuples besides the top-k ones that also satisfy the query but cannot be returned. The scoring function may be simple (e.g., based on a single attribute) or complex (e.g., computed from multiple attributes), static (i.e., independent of the search query) or dynamic, and may be known or unknown to the users.

Recently, there has been growing interest by third-party applications in obtaining *random samples* of the data in online hidden databases [12–14]. Such samples can be of great benefit to third-party applications, because various analytical tools can be enabled from the sample. For example, statistical information about the data can be derived, such as useful aggregates [12, 13]. A variety of sampling methods can be employed to produce the random samples - e.g., simple random sampling, stratified sampling, etc. For the purpose of this paper, we focus on simple random sampling which selects each tuple with equal probability. A justification for this choice is provided in Section 5. In the latter part of this paper, unless otherwise specified, our usage of term "sampling" refers specifically to simple (uniform) random sampling.

Our previous work [13] shows that, for hidden databases which provide the actual COUNT information (i.e., the total number of tuples that satisfy the given query) along with the top-k returned tuples, uniform random sampling can be done very efficiently. While a lot of hidden databases provide such COUNT information, numerous others do not or provide notoriously inaccurate COUNT information (e.g., Google). For these COUNT-less hidden databases, it has been challenging to develop effective sampling algorithms, primarily because complete and unrestricted access to the data is disallowed. We focus on the uniform random sampling of COUNT-less hidden databases in this paper, and summarize below the current state-of-the-art of such samplers.

1.1 Current State-of-the-Art Samplers

There are two main objectives that a sampling algorithm should seek to achieve:

- *Efficiency:* The efficiency of the sampling process is measured by the number of queries that need to be executed via the web 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.

- *Minimizing Skew:* 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 skew, i.e., the probability for each tuple to be selected should deviate as little as possible from the uniform distribution.

[*]Partly supported by NSF grants 0852673, 0852674, 0845644, and 0915834
[†]Partly supported by NSF grants 0845644, 0812601, 0915834, and grants from Microsoft Research and Nokia Research

A state-of-the-art sampler for hidden databases is the HIDDEN-DB-SAMPLER, proposed in [12]. This approach is based on a random drill-down process of queries executable via the form-like interface - it starts with an extremely broad (therefore overflowing) query, and iteratively narrowing it down by adding random predicates, until a non-overflowing query is reached. If such a query answer is not empty, one of the returned tuples is randomly picked for inclusion into the sample (subject to a probabilistic rejection test to reduce skew). Otherwise, the drill-down process restarts from the extremely broad query. This process can be repeated to get samples of any desired size.

However, HIDDEN-DB-SAMPLER and all other existing hidden database samplers (i.e., COUNT-DECISION-TREE [13] and HYBRID-SAMPLER [13]) have a main deficiency in that they *ignore all overflowing queries*. In the design of these samplers, tuples returned as the result of an overflowing query are assumed to be useless for assembling into a random sample because they have been selected *not* by a random procedure, but preferentially based on the pre-determined scoring function. Thus, overflowing queries are ignored in order to avoid favoring highly ranked tuples and thereby incurring high skew in the retrieved sample. While not using overflowing queries is a simple solution to eliminate the effect of scoring function on skew, it also significantly increases the number of queries required for sampling, thus adversely impacting efficiency. For example, during the random drill-down process, existing hidden database samplers have to execute a large number of overflowing queries before they encounter a non-overflowing one.

1.2 Main Technical Contributions

In this paper, one of our main contributions is to propose novel techniques that *leverage overflowing queries* to turbo-charge the efficiency of samplers. Thus, overflowing queries that were ignored by prior samplers can now be properly utilized during the sampling process. In particular, we develop **TURBO-DB-SAMPLER**, a sampler for hidden databases that is an order of magnitude (10 times in our experiments) more efficient than the existing HIDDEN-DB-SAMPLER. The main idea behind our approaches is the novel concept of a *designated query* that maps each tuple in the database to a unique query (whether overflowing or not) executable via the form-like interface, and a *designation test* procedure that efficiently determines whether a query is designated for a tuple.

We also find that, for hidden databases with static scoring functions (i.e., which are independent of the search queries), a simpler designation test is available which enables the sampling efficiency to be further improved by a novel scheme called *level-by-level sampling*. This scheme skips queries in the drill-down process whose results are not picked for inclusion into the sample. With this scheme, we develop **TURBO-DB-STATIC**, an algorithm that achieves an additional speedup factor of 2 for databases with static scoring functions.

The second main contribution of our paper is that our algorithms also significantly reduces sampling skew. In particular, we propose a novel scheme of *concatenating sampling with crawling* to reduce sampling skew. The basic idea is to switch to crawling from the random drill-down process if we encounter a query which remains overflow after adding a large number of predicates (i.e., a long overflowing query). The premise here is that a long overflowing query is an indication of a dense cluster of data tuples - over which the random drill-down technique produces an extremely high skew. The usage of crawling over such dense clusters significantly reduces the sampling skew while maintaining a low query cost because the number of queries needed for the crawling of such a cluster is linearly bounded by the number of tuples matching the long overflowing query, which is usually small. While this concatenated scheme is not able to completely remove skew, we show that TURBO-DB-SAMPLER incurs orders of magnitude smaller skew than HIDDEN-DB-SAMPLER and other state-of-the-art samplers.

The contributions of this paper may be summarized as follows:

- We develop a novel technique of using *designated queries* and *designation tests* to leverage overflowing queries and thereby dramatically improve the efficiency of sampling.
- We develop a novel technique of *concatenating sampling with crawling* to significantly reduce sampling skew.
- For hidden databases with static scoring functions, we develop a novel technique of *level-by-level sampling* to further improve sampling efficiency substantially.
- For hidden databases with arbitrary scoring functions, we put together the first two techniques to develop a generic **TURBO-DB-SAMPLER** which achieves a speedup factor of more than 5 over HIDDEN-DB-SAMPLER [12].
- For hidden databases with static scoring functions, we put together all three techniques to develop **TURBO-DB-STATIC** which further improves sampling efficiency by a speedup factor of 2.
- We run extensive experiments to demonstrate the effectiveness of our proposed sampling algorithms. In particular, we tested our results on both synthetic datasets and a real-world dataset crawled from Yahoo! Auto [22]. The experimental results demonstrate the superiority of our sampling algorithms over the previous efforts.

The rest of this paper is organized as follows: We define the problem and briefly review the existing samplers in Section 2. In Sections 3, we introduce TURBO-DB-SAMPLER, our major sampling algorithm, and its two main ideas: leveraging overflowing queries with designation tests, and reducing skew by concatenating sampling with crawling. In Section 4, we introduce TURBO-DB-STATIC, a sampling algorithm which leverages the static scoring function to enable a more efficient designation test and a level-by-level sampling scheme that further improves efficiency. We present related discussions in Section 5. Section 6 shows the experimental results. Related work is reviewed in Section 7, followed by final remarks in Section 8.

2. PRELIMINARIES

In this section, we introduce a model of hidden databases, define the performance measures for sampling over hidden databases, and review the state-of-the-art HIDDEN-DB-SAMPLER.

2.1 Model of Hidden Databases

Hidden databases on the web receive queries from users through web forms. To form a query, users are generally provided with drop down boxes, check boxes, text boxes or other common *HTML* form elements. After a query is selected by the user, a request is submitted and the hidden database system return tuples matching the user query. Large databases generally restrict users access to top-k tuples which may be presented on one pages or over multiple pages (accessed by page turns or clicking next at the bottom of the results page). Below, we provide a simple formalization of this model.

Consider a hidden database table T with n tuples $t_1, \ldots, t_n$ and m attributes $a_1, \ldots, a_m$ which have respective domains of $D_1, \ldots, D_m$.

We restrict our discussion in the most part of this paper to categorical data, and discuss the extension to numerical databases in Section 5.2. We point out that a large number of hidden database systems can be transformed into categorical data by applying simple transformations. As an example, consider a *price-range* attribute which provides upper and lower bounds. This can be used as a combination of discrete intervals with an upper and lower bound. Table 1 shows an example with $n = 4$, $m = 3$ and $D_i = \{0, 1\}$ ($i \in [1, 3]$). Suppose $k = 1$ and the four tuples ranked by score from high to low are $t_1, \ldots, t_4$. This table will be used throughout the paper as a running example.

Table 1: A Running Example for Hidden Databases

	a_1	a_2	a_3
t_1	0	0	0
t_2	0	0	1
t_3	1	0	0
t_4	1	1	0

2.1.1 *Model of Query Input and Output*

The hidden database table is only accessible to users through a web-based search interface which allows users to query the database by specifying values for a subset of attributes. Thus a user query Q_S is of the form: `SELECT` $*$ `FROM` T `WHERE` $a_{i_1} = v_{i_1} \& \ldots \& a_{i_s} = v_{i_s}$, where $\{a_{i_1}, \ldots, a_{i_s}\} \subseteq [0, m-1]$ and $v_{i_j} \in D_{i_j}$. For the running example, a user may specify a query `SELECT` $*$ `FROM` T `WHERE` $a_1 = 0$ `AND` $a_2 = 0$.

The query interface is restricted to only return k tuples, where k is a pre-determined small constant (such as 20 or 50). Thus, the tuples that match a query Q_S, $Sel(Q_S)$, 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 $Sel(Q_S)$ (according to a scoring function) will be returned as the query result. The interface will also notify the user that there is an *overflow,* i.e., that not all documents matching Q_S can be returned. For example, if $k = 1$, a query in the running example, Q_S: `SELECT` $*$ `FROM` T `WHERE` $a_1 = 0$ cannot be entirely returned because $Sel(Q_S) = \{t_1, t_2\}$ contains more than k tuples. As a result, t_1, t_2 have to be evaluated against the scoring function, and the one with the higher score will be returned along with an overflow notification.

At the other extreme, if the query is too specific and matches no tuple, we say that an *underflow* occurs. In the running example, Q_S: `SELECT` $*$ `FROM` T `WHERE` $a_1 = 1$ `AND` $a_3 = 1$ underflows. If there is neither overflow nor underflow, we have a *valid* query result. An example of valid query is Q_S: `SELECT` $*$ `FROM` T `WHERE` $a_1 = 0$ `AND` $a_3 = 1$ in the running example. In this paper, we assume the query answering system to be deterministic, i.e., the same query executed again will produce the same set of results.

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 the search conditions. We argue that this is a reasonable assumption because many real-world top-k interfaces (e.g., Google Base [16], Yahoo Auto [22]) only allow "page turns" for limited (100) times.

Since the tuples returned by a query may not include tuples that match the query, for the purpose of clarification, we distinguish the meaning of three verbs: *match*, *return*, and *draw*, which are extensively used in the paper. We say a tuple *matches* a query iff the tuple is in the hidden database and satisfies the selection condition of the query. The number of tuples that match a query may be greater than k. We say a tuple is *returned* by a query iff the tuple is one of those that are actually displayed on the query output interface as the response to that query. The number of tuples that a query can return is always smaller than or equal to k. While a tuple returned by a query always matches the query, the reverse is not always true. We say a tuple is *drawn* from a query during the sampling process iff the sampler selects the tuple from the returned answer of the query, and then uses that tuple as a sample tuple. A sample tuple drawn from a query must be returned by the query and therefore matches the query. Again, the reverse is not always true.

2.1.2 *Model of Scoring Function*

There are two types of scoring functions for a hidden database: One is *static* in that each tuple's score is a function of the tuple value, and does not change with different user-issued search queries. For example, a real estate database may score each tuple (i.e., house) by its price (i.e., an attribute value), and only display the k cheapest houses. The scoring function we use for the running example is also static. The other type of scoring function is *query-dependent* and is a function of both the tuple value and the search query. For example, the real estate database may score each tuple by price if price is not included in the search conditions, or by the number of bedrooms if price is included. In this paper, unless otherwise specified (e.g., in Section 4 which focuses on static ranking functions), we consider generic ranking functions which may be static or query-dependent.

2.2 Performance Measures

Recall from the introduction that our objective is to generate a uniform random sample over the hidden web database. Such a sampling algorithm should be measured in terms of efficiency and skew which we formally define as follows.

- Efficiency: For a given (desired) sample size s, we measure the efficiency of a sampler by the expected number of queries it issues to the form-like web interface in order to obtain s sample tuples.
- Skew: We define the *level of skew* as the standard deviation of the probability for a tuple to be sampled. i.e.,

$$\gamma = \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} \left(\Pr\{t_i \text{ is chosen as the sample}\} - \frac{1}{n} \right)^2} \quad (1)$$

We observe later that the above measures tradeoff against each other. The parameter of our system enables the user to strike the right balance according to their specific needs.

2.3 HIDDEN-DB-SAMPLER

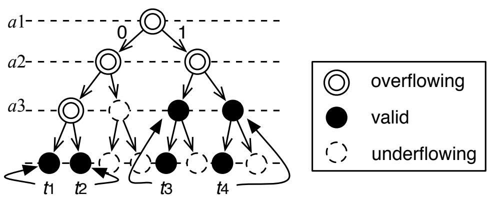

Figure 1: HIDDEN-DB-SAMPLER for the Running Example

We now review HIDDEN-DB-SAMPLER, the sampling algorithm presented in our earlier work [12] for obtaining random samples from hidden databases. Consider a tree constructed from an arbitrary order of all attributes, say $a_1, \ldots, a_m$. Let the root be the Level 1. All internal nodes of the tree at the ith level are labeled by attribute a_i. Each internal node a_i has exactly $|D_i|$ edges leading out of it, labeled with values from D_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. Figure 1 depicts such a query tree for a Boolean database. This tree will be used as a running example throughout the paper. 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 underflowing.

HIDDEN-DB-SAMPLER issues queries from the tree to obtain a random sample tuple. To simplify the discussion, assume $k = 1$. Suppose we have reached the i-th level (suppose that the root level is Level-0) and the path thus far represents the query $a_1 = v_1 \& \ldots \& a_i = v_i$.. The algorithm selects one of the domain values of a_{i+1} uniformly at random, say v_{i+1}, adds the condition $a_{i+1} = v_{i+1}$ 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 skew into the sample, as not all tuples are reached with the same probability. Techniques such as acceptance/rejection sampling are further employed for reducing skew (see [12] for further details).

3. TURBO-DB-SAMPLER

In this section, we describe TURBO-DB-SAMPLER, our algorithm for sampling a hidden database. TURBO-DB-SAMPLER features two main ideas: it leverages overflowing queries to significantly improve the efficiency of sampling; and concatenate sampling with crawling to substantially reduce the skew of samples. In the following, we first develop two samplers: OVERFLOW-SAMPLER and CONCATENATE-SAMPLER with the two respectively, and then combine them together to construct TURBO-DB-SAMPLER.

3.1 Improve Efficiency: Leverage Overflows

3.1.1 Basic Idea

We leverage overflowing queries by ignoring the overflowing flag and treating an overflowing query in the same way as a valid query that returns k tuples. However, doing so makes a tuple with a higher score to be returned by more queries, and to have a higher probability of being selected into the sample. To eliminate such score-related skew, we place a restriction on the selection of a tuple into the sample. In particular, we define one *designated query* (in the set of all interface-executable queries) for each tuple in the hidden database, and require each selected sample tuple to go through a *designation test* - i.e., we enforce a rule that a tuple can be selected into the sample *only if* it is retrieved from the result of its designated query. Note that the designated query for a tuple can be either valid or overflowing.

To understand the effect of this change on hidden database sampling, let us first investigate a simple application of it to the random drill-down process of HIDDEN-DB-SAMPLER. At this moment, let us assume that the designation test can be done without issuing any additional query to the web interface. We will discuss details about the cost of designation test in the next step. There are two possible outcomes of this change for the generation of one sample tuple:

- First, the random drill-down process might select a sample tuple from an overflowing query, leading to an earlier termination of the random drill-down process. This reduces the number of queries that need to be issued.

- However, depending on the definition of designated query, there might be a second outcome where the sampler has to drill deeper down the tree than HIDDEN-DB-SAMPLER if, according to the designated query definition, none of the overflowing and valid queries issued on upper levels are the designated queries for the tuples they return.

One can see from the two outcomes that the impact of this change on the efficiency of sampling is determined by the definition of designated queries. Indeed, if the designated query of a tuple is defined to be the highest-level *valid* query that returns the tuple, then HIDDEN-DB-SAMPLER remains the same after applying the change because no overflowing query will serve as the designated query for a tuple anyway.

To improve sampling efficiency, we define the designated query of a tuple as follows:

DEFINITION 3.1. *For a given query tree, the designated query of a tuple t is the highest-level query (valid or overflowing) that returns t. We denote the designated query by $q(t)$.*

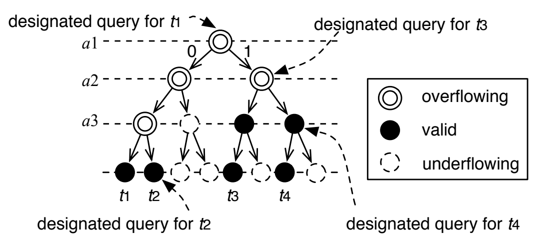

Figure 2: Designated Queries in the Running Example

Figure 2 shows the designated queries of all four tuples for the running example in Table 1 (recall that $k = 1$ in the example). One can see from the figure that, compared with HIDDEN-DB-SAMPLER, t_1 and t_3 may be retrieved by higher-level *overflowing* queries. There are two important observations from this definition of designated queries:

- First, note that for a given tuple t, at any given level, there is at most one query which returns t. Thus, the definition guarantees each tuple to have one and only one designated query. This observations ensures that a tuple with higher score will not be selected with higher probability after leveraging overflowing queries.

- Second, if the drill-down process is used, the designation test can be done without issuing any additional queries. To understand why, consider the designation test for a query Q and a tuple t which is selected from the result of Q. When the drill-down process reaches Q, it must have issued all queries on the path between the root node and Q, which include all possible higher-level queries that may return t. Thus, the sampler can just look up the answers of these queries from the historic log, and then pass the designation test iff t was never returned in these queries.

3.1.2 Algorithm OVERFLOW-SAMPLER

We now describe OVERFLOW-SAMPLER, the sampling algorithm which uses our definition of designated queries. Similar to HIDDEN-DB-SAMPLER, OVERFLOW-SAMPLER uses C to be the cut-off level for balancing between efficiency and skew. Since most overflowing queries occur on higher levels of the tree, for the purpose of this subsection, we focus on the sampling of such tuples on or above Level-C, and defer the discussion of sampling tuples below Level-C to the next subsection which specifically addresses this issue. Therefore, we assume that no tuple is hidden below Level-C - i.e., all Level-C queries are valid or underflowing.

OVERFLOW-SAMPLER again uses a random drill-down process starting from the root level of a given query tree. However, before drilling down the tree, the sampler first determines whether the sample tuple can be drawn from the root query. In particular, it randomly chooses a tuple from those returned by the root query, and selects it as a sample with probability of

$$p_0 = \frac{|d(Q_0)|}{k \cdot \pi(C)}, \tag{2}$$

where $|d(Q_0)|$ is the number of tuples that have the root query as their designated query, and function $\pi(\cdot)$ is defined as

$$\pi(i) = \prod_{j=1}^{i} |D_j| \tag{3}$$

which is the product of the domain size of attributes a_1 to a_i. Let $\pi(0) = 1$. For example, the root query in our running example returns t_1 with probability of $1/2^3$. Note that according to our definition of the designated query, a tuple has the root as its designated query iff it is returned by the root query. As such, $d(Q_0) = |Q_0|$, the number of tuples *returned* by Q_0. Therefore, OVERFLOW-SAMPLER selects each tuple returned by the root with probability of

$$\frac{1}{|Q_0|} \cdot p_0 = \frac{1}{|Q_0|} \cdot \frac{|Q_0|}{k \cdot \pi(C)} = \frac{1}{k \cdot \pi(C)}. \tag{4}$$

If a sample tuple is not selected from the root *and* the root is overflowing, then OVERFLOW-SAMPLER starts the random drill-down process. Note that if the root is valid or underflowing, then no query with additional predicates can serve as the designated query for any tuple. Thus, the sampling process has to restart in this case.

Different from HIDDEN-DB-SAMPLER, in the drill-down process, OVERFLOW-SAMPLER follows each outgoing branch with *different* probability. The purpose of doing so is to address the fact that tuples returned from upper levels of the tree may also satisfy the outgoing branches, but will not be returned by any lower level queries. Therefore, these tuples must be excluded from consideration in the computation of outgoing probabilities. In particular, OVERFLOW-SAMPLER follows an outgoing branch $a_i = v_i$ (of a Level-$(i-1)$ node Q_{i-1}) with probability of

$$\beta_i(v_i) = \frac{\pi(i-1)}{\pi(i)} \cdot \frac{k \cdot \pi(C) - |f(Q_{i-1}, v)| \cdot \pi(i)}{k \cdot \pi(C) - |f(Q_{i-1})| \cdot \pi(i-1)}, \tag{5}$$

where $f(Q_{i-1})$ is the set of tuples returned by $Q_0, \ldots, Q_{i-1}$ which satisfy Q_{i-1}, and $f(Q_{i-1}, v_i)$ is the subset of $f(Q_{i-1})$ which further satisfies $a_i = v_i$. In the running example, we have $\beta_0(0) = (1/2)\cdot(2^3-2)/(2^3-1) = 3/7$ and $\beta_0(1) = (1/2)\cdot 2^3/(2^3-1) = 4/7$. Thus, the sampler follows the left and right branches of the root node with probability of $3/7$ and $4/7$, respectively. This is consistent with the intuition that, since the tuple returned by the root "belongs to" the left branch, the drill-down process should turn to the left with lower probability.

During the drill down process, after issuing each query, the sampler again needs to determine whether the sample tuple can be drawn from the results. Consider such a process for a Level-i query Q_i. Let the list of queries issued so far by the drill down process be $Q_0, \ldots, Q_{i-1}$, at levels 0 to $i-1$, respectively. The sampler first computes $d(Q_i)$ as

$$d(Q_i) = Q_i \backslash (Q_0 \cup \cdots \cup Q_{i-1}). \tag{6}$$

For instance, in the running example, query with predicates $(a_1 = 0)\&(a_2 = 0)$ has $d(Q) = t_1 \backslash t_1 = \varnothing$. If $d(Q_i)$ is not empty, the sampler randomly draws a tuple from $d(Q_i)$ and selects it as a sample with probability of

$$p_i = \frac{|d(Q_i)|}{k \cdot \pi(C) \cdot (\prod_{j=1}^{i-1} \beta_j(v_j)) \cdot \prod_{j=1}^{i-1}(1-p_j)}. \tag{7}$$

We can derive the value of p_i from this iterative formula and the value of p_0 in (2):

$$p_i = \frac{|d(Q_i)| \cdot \pi(i)}{k \cdot \pi(C) - |f(Q_{i-1}, v_i)| \cdot \pi(i)}. \tag{8}$$

In the running example, when query $(a_1 = 1)$ is issued, the sampler has probability of $2^1/(2^3 - 0) = 1/4$ to select t_3 as the sample. Note that there is always $p_i \leq 1$. To understand why, note that p_i increases monotonically with i. When $i = C$, we have

$$p_C = \frac{|d(Q_C)|}{k - |f(Q_{C-1}, v_C)|}, \tag{9}$$

Since $d(Q_C)$ and $f(Q_{C-1}, v_C)$ are mutually exclusive sets of tuples that satisfy Q_C, according to our assumption for all Level-C queries to be valid or underflowing, there must be $|d(Q_C)| + |f(Q_{C-1}, v_C)| \leq k$. Therefore, $p_C \leq 1$.

From the definition of p_i, one can derive that the probability for a tuple with designated query Q_i to be selected as a sample is

$$\left(\prod_{j=1}^{i} \beta_j(v_j)\right) \cdot \left(\prod_{j=0}^{i-1}(1-p_j)\right) \cdot \frac{1}{|d(Q_j)|} \cdot p_i = \frac{1}{k \cdot \pi(C)} \tag{10}$$

which is constant across all levels. Again, if no sample is drawn and Q_i overflows, the drill down process continues to deeper levels. Otherwise, the sampler either terminates (if enough sample has been collected) or restarts from the root.

Algorithm 1 depicts the pseudocode for OVERFLOW-SAMPLER. For setting the value of the parameter C, we follow the heuristic rule in [12] that C be set as the average depth of a random walk. In the running example, this leads to an assignment of $C = (0+1+2+3)/4 = 1.5$. Note that in practice, a sampler has no prior knowledge of the average depth. Nonetheless, as discussed in [12], the value of C can be determined adaptively during the sampling process because the average depth can be learned as more and more random walks are accomplished.

3.1.3 Analysis of Improvement on Efficiency

We now discuss the efficiency comparison between OVERFLOW-SAMPLER and HIDDEN-DB-SAMPLER. First, one can observe that, to achieve the same level of skew, the number of queries required by OVERFLOW-SAMPLER never exceeds that of HIDDEN-DB-SAMPLER. To understand why, consider the execution of both samplers with the same value of C over the same hidden database.

Algorithm 1 OVERFLOW-SAMPLER for Levels 1 to C

```
Require: $C$, a pre-determined cut-off level
 1: $Q \leftarrow$ SELECT * FROM D, $prod \leftarrow 1, i \leftarrow 0$.
 2: Issue query $Q$, Compute $d(Q)$.
 3: $p \leftarrow |d(Q)| \cdot \pi(i)/(k \cdot \pi(C) \cdot prod)$.
 4: Randomly generate $r \in (0,1)$ according to uniform distribution.
 5: if $r \leq p$ then
 6:     Randomly choose a tuple from $d(Q)$ as sample. exit.
 7: end if
 8: if $Q$ overflows then
 9:     $prod \leftarrow prod \cdot (1-p)$. $i \leftarrow i+1$.
10:     Randomly generate $v \in D_i$. Add predicate $a_i = v$ to $Q$.
11:     Goto 2
12: else
13:     Goto 1
14: end if
```

As we have shown, for both algorithms, a random drill-down process draws each tuple on or above Level-C with probability of $1/(k \cdot \pi(C))$. However, for any tuple, the number of queries (i.e., levels to step down) required by OVERFLOW-SAMPLER to draw the tuple is always smaller than or equal to that of HIDDEN-DB-SAMPLER. Thus, OVERFLOW-SAMPLER can only improve the efficiency of sampling for a given level of skew.

In the following, we quantitatively analyze the efficiency improvement provided by OVERFLOW-SAMPLER. Since OVERFLOW-SAMPLER is only used for sampling tuples on or above Level-C, the ratio of improvement depends on the value of C. As previously discussed, we set the value of C to be the average depth of a random walk for both algorithms. Similar theorems can be derived for other settings of C.

THEOREM 3.1. *For a given query tree with u_{O} overflowing nodes, n tuples, and a top-k interface, the ratio between the query cost of HIDDEN-DB-SAMPLER and OVERFLOW-SAMPLER on obtaining one sample tuple is*

$$\frac{E(\textit{cost of HIDDEN-DB-SAMPLER})}{E(\textit{cost of OVERFLOW-SAMPLER})} \geq 2^{k \cdot u_{\mathrm{O}}/n}. \quad (11)$$

We omit the proof due to space limitation. The theorem follows directly from the following observation: If a tuple appears in d overflowing queries, then the random walk to reach this tuple in HIDDEN-DB-SAMPLER is d levels longer than in OVERFLOW-SAMPLER. Thus, on average the length of a random walk that draws a sample tuple in OVERFLOW-SAMPLER is $k \cdot u_{\mathrm{O}}/n$ shorter than that in HIDDEN-DB-SAMPLER. In the running example, such average length is $(0+1+2+3)/4 = 1.5$ for OVERFLOW-SAMPLER and $(2+2+3+3)/4 = 2.5$ for HIDDEN-DB-SAMPLER, leading to a difference of exactly $k \cdot u_{\mathrm{O}}/n = 1 \cdot 4/4 = 1$. Since each attribute has at least two values, $2^{k \cdot u_{\mathrm{O}}/n}$ serves as a lower bound for the improvement ratio.

The theorem indicates that OVERFLOW-SAMPLER can achieve a significant improvement ratio in practice. For example, consider a 50-attribute, $10,000$-tuple, Boolean i.i.d dataset with probability of 1 being 0.1 for each attribute and $k = 1$. There are an expected number of $31,176.95$ overflowing queries in the query tree. According to the theorem, by leveraging overflowing queries one can reduce the expected query cost by a factor of at least 8.68.

3.2 Reduce skew: Concatenate with Crawling

3.2.1 Basic Idea

As we mentioned in the previous subsection, both HIDDEN-DB-SAMPLER and OVERFLOW-SAMPLER can produce unbiased samples if all Level-C queries are valid. Thus, sampling skew is caused by the different selection probability of tuples with designated queries below the cutoff Level-C. For example, if a tuple is only returned by a leaf-level query but not higher-level ones (e.g., due to the tuple's low score), then HIDDEN-DB-SAMPLER selects this tuple with probability only $\pi(C)/\pi(m) \leq 1/2^{m-C}$ times of that for a tuple returned on or above Level-C.

To reduce the skew for sampling these low-level tuples, our main idea is to *concatenate sampling with crawling* on levels below Level-C. In particular, the sampling of tuples below Level-C starts when OVERFLOW-SAMPLER reaches an overflowing query Q at Level-C but could not select a sample from it (due to rejection sampling in Line 5 of Algorithm 1). Instead of further conducting the random walk to drill below Level-C, we switch to the crawling of the subtree of Q, and compute $\Omega(Q)$, the set of all tuples in the database that *match* Q. Then, we randomly choose a tuple from $\Omega(Q)$ and return it as a sample after a rejection sampling step which will be discussed in detail later. The crawling process can be performed as a depth-first search of the subtree with root being Q. The depth-first search backtracks from a node if the node is already valid or underflowing.

The concatenation of sampling with crawling has two main implications on the performance of sampling:

- The concatenated process substantially reduces the skew of selection probability for tuples below Level-C. Unlike in HIDDEN-DB-SAMPLER where a tuple's selection probability decreases exponentially with its level index below Level-C, the concatenation idea ensures equal selection probability for all (lower-level) tuples in the subtree of an overflowing Level-C node. As we shall show at the end of this subsection, the difference between the selection probability of different tuples is substantially reduced with the concatenated process.

- The effect of crawling on sampling efficiency is insignificant for practical databases. The crawling of a subtree in which n_{T} tuples have designated queries below Level C requires $O(n_{\mathrm{T}})$ queries. In the theoretical worst case, the subtree may contain as many as $n_{\mathrm{T}} = \max(n-k, k \cdot \pi(m)/\pi(C) - k)$ tuples which leads to an unacceptably large query cost. Nonetheless, we argue that this case is extremely unlikely to occur in practice because it indicates that attributes $a_1, \ldots, a_C$ (in the upper levels) are incapable of distinguishing tuples, and should be excluded from the form-like web interface. For real-world datasets, when C is reasonably large (e.g., $k \cdot \pi(C) \approx n$), the number of tuples in each subtree should be fairly small.

3.2.2 Algorithm CONCATENATE-SAMPLER

We now describe the detailed design of CONCATENATE-SAMPLER, our algorithm for sampling tuples with designated queries below Level-C by concatenating sampling with crawling.

Recall from the above subsection that, when all Level-C queries are valid or underflowing, a random drill-down process of OVERFLOW-SAMPLER selects each tuple with probability $\zeta = 1/(k \cdot \pi(C))$. When there exists tuples below Level-C (i.e., there exists Level-C overflowing queries), however, it is no longer possible to always select each tuple with probability of $1/(k \cdot \pi(C))$. To understand why, consider an extreme case where all Level-C queries overflow.

In this case, the number of tuples $n > k \cdot \pi(C)$, making it impossible to select each tuple with probability of $1/(k \cdot \pi(C))$.

Thus, in order to support the sampling of tuples below Level-C, we have to decrease the value of ζ. In particular, we consider a new (target) selection probability

$$\zeta_{\mathrm{I}} = \frac{1}{(k + n_0) \cdot \pi(C)}, \tag{12}$$

where n_0 is the maximum number of tuples that *match* a Level-C overflowing query. Although the sampler has no prior knowledge of n_0, it can learn it adaptively when more Level-C subtrees are crawled. The adaption of OVERFLOW-SAMPLER to this new selection probability can be easily done by replacing k in Algorithm 1 with $k + n_0$. Thus, in the following, we focus on the sampling of tuples with designated queries below Level-C with selection probability as close to ζ_{I} as possible (i.e., to achieve minimum skew).

Algorithm 2 CONCATENATE-SAMPLER for Levels $C + 1$ to m

```
1: Crawl the subtree of $Q$. Suppose that $n_{\mathrm{T}}$ tuples with designated queries below Level $C$ are collected.
2: With probability of $n_{\mathrm{T}}/n_0$, return a tuple randomly chosen from the $n_{\mathrm{T}}$ tuples.
3: Update the value of $n_0$ if necessary.
```

Let Q_C be a Level-C overflowing query. After adapting OVERFLOW-SAMPLER to the new ζ_{I}, a tuple randomly drawn from the returned result of Q_C is selected as a sample with probability of $p_C = |d(Q_i)|/(k + n_0 - |f(Q_{C-1}, v_C)|)$. If it is not selected as a sample, our CONCATENATE-SAMPLER will be invoked with probability of $n_0/(k + n_0 - |f(Q_C)|)$. This way, given Q_C, the overall probability for a random-drill down process to switch to the crawling of the subtree of Q_{C} is $n_0/((k + n_0) \cdot \pi(C))$.

Once the CONCATENATE-SAMPLER is invoked, we crawl the subtree of Q_{C} and randomly draw from the crawling result a tuple t with designated query below Level C. Suppose that the subtree includes n_{T} such tuples. Then, with probability of $\max(n_{\mathrm{T}}, n_0)/n_0$, we select t as a sample tuple. Otherwise, CONCATENATE-SAMPLER is aborted and OVERFLOW-SAMPLER restarts from the root level.

Algorithm 2 depicts pseudocode of CONCATENATE-SAMPLER.

3.2.3 Analysis of Reduction on Skew

We now discuss the skew comparison between CONCATENATE-SAMPLER and HIDDEN-DB-SAMPLER. One can observe that CONCATENATE-SAMPLER can still be skewed - if an overflowing query at Level C has a subtree with size more than (the current value of) n_0, then tuples with designated queries in this tree will be sampled with lower probability than the others. In particular, a tuple in a n_{T}-tuple subtree will be selected with probability $\min(1, n_0/n_{\mathrm{T}})$ times of that for a tuple on or above Level C.

However, as we argued in the beginning of this subsection, with a reasonable value of C (e.g., $\pi(C) \approx n$), we expect $n_{\mathrm{T}} \leq n_0$ to hold true for most subtrees. Even when $n_{\mathrm{T}} > n_0$, the ratio between the highest and lowest selection probability for different tuples is at most n/n_0, which in many cases is still much smaller than the $\pi(m)/\pi(C)$ ratio for HIDDEN-DB-SAMPLER, because for most hidden databases the size of the database n is orders of magnitude smaller than the space of all possible tuple values ($\pi(m)$).

The following theorem further investigates the skew comparison for an i.i.d. Boolean dataset with probability of 1 being 0.5 and $k = 1$.

THEOREM 3.2. *When $2^C \gg n$ and m is sufficiently large, the level of skew of HIDDEN-DB-SAMPLER γ_{H} and that of TURBO-DB-SAMPLER γ_{T} satisfies*

$$\frac{\gamma_{\mathrm{T}}}{\gamma_{\mathrm{H}}} < 2^{C/2+1} \cdot \left(\frac{n-1}{2^C}\right)^{n_0/2} \tag{13}$$

which is much smaller than 1 given a reasonably large n_0.

We omit the proof due to space limitation. One can see from the theorem that the sample skew is significantly smaller in CONCATENATE-SAMPLER than HIDDEN-DB-SAMPLER. For example, given a Boolean database with 10,000 tuples, 100 attributes, and $C = 20$, when $n_0 = 5$, CONCATENATE-SAMPLER has a skew of 0.018 times that of TURBO-DB-SAMPLER. The skew ratio will further reduce for a larger value of n_0.

Since the adoption of CONCATENATE-SAMPLER also incurs query cost on the crawling of low-level tuples, we analyze the expected query cost incurred by CONCATENATE-SAMPLER. In particular, we integrate the lower-level sampling of CONCATENATE-SAMPLER with the higher-level sampling of HIDDEN-DB-SAMPLER, and then compare it with a pure HIDDEN-DB-SAMPLER which uses the random walk theme throughout all levels. The comparison is analyzed in the following theorem.

THEOREM 3.3. *If n and m are sufficiently large, when a Level-C subtree contains at most n_0 tuples, to achieve the same level of skew, the ratio between the query cost of HIDDEN-DB-SAMPLER and CONCATENATE-SAMPLER on obtaining one sample is*

$$\frac{E(\textit{cost of HIDDEN-DB-SAMPLER})}{E(\textit{cost of CONCATENATE-SAMPLER})} \geq \frac{n_0 \cdot \pi(C) \cdot k}{(k + n_0) \cdot \pi(C) + n_0 \cdot n/2} \tag{14}$$

Again, we omit the proof due to space limitation. One can see from the theorem that despite of requiring the crawling of a subtree, CONCATENATE-SAMPLER actually requires substantially fewer queries than HIDDEN-DB-SAMPLER to achieve the same level of skew. In the above Boolean database example with $n = 10,000, m = 100$, and $C = 20$, when $n_0 = 5$, the theorem indicates that the cost of HIDDEN-DB-SAMPLER is at least 16.60 times that of CONCATENATE-SAMPLER.

3.3 Algorithm TURBO-DB-SAMPLER

Algorithm 3 depicts the pseudocode for TURBO-DB-SAMPLER.

4. TURBO-CHARGING SAMPLERS FOR STATIC SCORING FUNCTIONS

In this section, we describe TURBO-DB-STATIC, our algorithm for sampling a hidden database with a static scoring function. TURBO-DB-STATIC integrates the two ideas we discussed for TURBO-DB-SAMPLER (i.e., leveraging overflows and concatenating sampling with crawling), as well as a third idea of level-by-level sampling which is enabled by a unique property of static scoring functions and is capable of further improving the efficiency of sampling. We discuss this idea in this section.

4.1 Improve Efficiency: Level-by-Level Sampling

4.1.1 Basic Idea

We first describe the basic idea of level-by-level sampling, and then explain why it only applies to hidden databases with static scoring functions.

Algorithm 3 TURBO-DB-SAMPLER

```
Require: Parameters $C$ and $n_0$ set adaptively during sampling
 1: $Q \leftarrow$ SELECT * FROM D, $prod \leftarrow 1, i \leftarrow 0$.
 2: Issue query $Q$, Compute $d(Q)$.
 3: $p \leftarrow |d(Q)| \cdot \pi(i)/((k+n_0) \cdot \pi(C) \cdot prod)$.
 4: Randomly generate $r \in (0, 1)$.
 5: if $r \leq p$ then
 6:     Randomly choose a tuple from $d(Q)$ as sample. exit.
 7: end if
 8: if $Q$ overflows then
 9:     $prod \leftarrow prod \cdot (1-p)$. $i \leftarrow i+1$.
10:     Randomly generate $v \in D_i$. Add predicate $a_i = v$ to $Q$.
11:     Goto 2 if $i \leq C$, Goto 15 otherwise.
12: else
13:     Goto 1
14: end if
15: Crawl the subtree of $Q$. Suppose that $n_\mathrm{T}$ tuples with designated queries below Level $C$ are collected.
16: With probability of $n_\mathrm{T}/n_0$, return a tuple randomly chosen from the $n_\mathrm{T}$ tuples.
```

Similar to OVERFLOW-SAMPLER, the objective of level-by-level sampling is to further improve the efficiency for sampling tuples with designated queries on or above the cutoff level C. For sampling these tuples, both OVERFLOW-SAMPLER and HIDDEN-DB-SAMPLER perform random walks and terminates a random walk when (1) one of the tuples the current query returns is selected as a sample, and/or (2) the random walk reaches a valid or underflowing query. However, such a technique has the following efficiency problem: Almost all top-level queries will be issued because every random walk initiates from there. Although the number of these queries is small in comparison with the total number of possible queries and/or the size of the database, the cost of issuing them may be significant for drawing a small number of samples. Nonetheless, we argue that issuing these top-level queries are hardly useful for sampling due to the following two reasons:

- Tuples with top-level designated queries form only a very small fraction of the database. Thus, these top-level queries have slim chances of actually contributing a sample tuple drawn from their returned answers.
- The role of top-level queries on the "early termination" of a random walk is also doubtful: Unless a database is extremely skewed, it is unlikely that these high-level queries can be underflowing or valid.

To address this problem, the main idea of level-by-level sampling is to perform the sampling successively for each level of the query tree, with the order downward from the root, and to issue a *growing* number of queries per level during the step-down process. If a query Q is not underflowing, we perform designation tests to find the set of tuples with designated queries being Q, and then randomly choose a tuple t from these tuples. We select t as a sample after subjecting it to a rejection sampling test. The sampling process terminates when a sample tuple is selected. If no tuple is chosen on or above Level C, we randomly choose an overflowing query from Level-C and execute CONCATENATE-SAMPLER over it. If CONCATENATE-SAMPLER cannot select a sample (i.e., due to rejection sampling based on the subtree size), we again randomly choose a Level-C overflowing query and repeatedly execute CONCATENATE-SAMPLER until a sample tuple is selected.

As we shall show in the detailed algorithm, level-by-level sampling has two main features that contribute to the improvement of sampling efficiency:

- Level-by-level sampling issues top-level queries with extremely small probability due to their unlikeliness of being selected to contribute a sample tuple.
- Unlike the random walk process which often aborts without returning a sample, level-by-level sampling *always* return a sample from each step-down process.

Finally, note that level-by-level sampling only applies to static scoring functions due to the requirement of designation test. Unlike OVERFLOW-SAMPLER, level-by-level sampling does not have a drill-down process. Therefore, when a query Q is issued, queries on the path between the root and Q may not have been issued before. As a result, the designation test cannot be performed based on the historic query answers as in OVERFLOW-SAMPLER. A simple method to perform this test is to actually issue every query on the path. However, doing so leads to significant query cost and is contradictory to our idea of avoid issuing top-level queries.

On the other hand, it is possible to perform the designation test efficiently when the hidden database has a *static* scoring function. In this case, only a single query needs to be issued for designation test: To check whether a given query Q is the designated query of a tuple t that it returns, we only need to check whether the parent of Q returns t. If it does not, then Q is the designated query of t, and vice versa. In the running example, to determine whether $(a_1 = 1)\&(a_2 = 1)$ is the designated query for t_4, we only need to judge whether query $a_1 = 1$ returns t_4, and do not need to issue the root query. In general, the reason is that if t is not returned by the parent query, it certainly cannot be returned by upper-level queries due to the static nature of the scoring function. Therefore, t must have Q as its designated query.

4.1.2 Algorithm LEVEL-SAMPLER

Algorithm 4 depicts LEVEL-SAMPLER, our baseline algorithm for level-by-level sampling. The following discussion consists of two parts: First, we make a few remarks on applying LEVEL-SAMPLER in practice. Then, we show that it samples each tuple with designated queries on or above Level C with equal probability.

One can see from Algorithm 4 that LEVEL-SAMPLER requires the knowledge of n, the number of tuples in the database. We argue that such knowledge is usually available in practice, as many hidden database providers publicize their database size as an advertisement for usability. If the knowledge of n is not available, an upper-bound estimate of it can be used. In this case, all tuples with designated query on or above Level C will still be sampled with equal probability, but the probability will be different from those tuples below Level C. Nonetheless, we shall demonstrate in the experimental results that the skew is extremely small even with an inaccurate estimation of n, and much smaller than the skew of the existing algorithms.

Another note of caution from Algorithm 4 is that the cutoff level C must be chosen such that $k \cdot \pi(C) \leq n$. Since we would like to sample as many tuples by LEVEL-SAMPLER as possible, to avoid the skew of CONCATENATE-SAMPLER, a natural choice is to set C to be the maximum value that satisfies $k \cdot \pi(C) \leq n$. In Section 4.2, we shall show how the value of C can be further increased by pruning the query tree.

We now show that LEVEL-SAMPLER is equivalent with OVERFLOW-SAMPLER (for static scoring functions) by proving that it generates unbiased samples when all Level-C queries are valid or underflowing.

Algorithm 4 LEVEL-SAMPLER for Levels 1 to C

```
1: $h = 0$.
2: With probability of $(1 - k \cdot \pi(h)/n)$, Goto 8. Otherwise, randomly choose a query $Q$ from Level-$h$.
3: Goto 10 if $Q$ is empty. Otherwise randomly pick a tuple $t$ from the result of $Q$.
4: Generate $r \in (0, 1)$ uniformly at random.
5: if $r \leq |Q|/k$ and $Q$ is the designated query for $t$ then
6:     return $t$ as sample and exit.
7: else if $r \leq |Q|/k$ and $Q$ is not the designated query for $t$ then
8:     With probability of $k \cdot \pi(h)/n$, Goto 2.
9: end if
10: if $h \leq C - 1$ then
11:     Set $h = h + 1$, goto 2
12: else exit
13: end if
```

THEOREM 4.1. *LEVEL-SAMPLER returns each tuple with designated query on or above Level C with probability of* $1/n$.

PROOF. We prove the unskewedness of LEVEL-SAMPLER in two steps: First, we show that once LEVEL-SAMPLER reaches Level h ($h \in [1, C]$), each tuple with designated query at Level h has probability of $1/(n - n_{h-1})$ to be returned as a example, where n_{h-1} is the number of nonempty (i.e., valid and overflowing) nodes at Level $h - 1$ (for $h = 0$, we assume that $n_{-1} = 0$). Then, we show that the overall probability for each tuple to be returned is $1/n$.

Note from Algorithm 4 that once LEVEL-SAMPLER reaches Level h, the probability for LEVEL-SAMPLER to not return a tuple from Level h is

$$p(h) = 1 - \frac{k \cdot \pi(h)}{n} + \frac{k \cdot \pi(h)}{n} \cdot \left(\frac{k \cdot \pi(h) - n_h}{k \cdot \pi(h)} + \frac{n_{h-1}}{k \cdot \pi(h)} \cdot \left(1 - \frac{k \cdot \pi(h)}{n}\right) + \frac{n_{h-1}}{k \cdot \pi(h)} \cdot \frac{k \cdot \pi(h)}{n} \cdot \left(\frac{k \cdot \pi(h) - n_h}{k \cdot \pi(h)} + \frac{n_{h-1}}{k \cdot \pi(h)} \cdot \left(1 - \frac{k \cdot \pi(h)}{n}\right)\right) + \left(\frac{n_{h-1}}{k \cdot \pi(h)} \cdot \frac{k \cdot \pi(h)}{n}\right)^2 \cdot \right. \tag{15}$$

$$\left.\left(\frac{k \cdot \pi(h) - n_h}{k \cdot \pi(h)} + \frac{n_{h-1}}{k \cdot \pi(h)} \cdot \left(1 - \frac{k \cdot \pi(h)}{n}\right)\right) + \cdots\right)$$

$$= 1 - \frac{k \cdot \pi(h)}{n} + \frac{k \cdot \pi(h)}{n} \cdot \frac{1 - \frac{n_h - n_{h-1}}{k \cdot \pi(h)} - \frac{n_{h-1}}{n}}{1 - \frac{n_{h-1}}{n}} \tag{16}$$

$$= 1 - \frac{k \cdot \pi(h)}{n} + \frac{k \cdot \pi(h)}{n} - \frac{n_h - n_{h-1}}{n - n_{h-1}} \tag{17}$$

$$= \frac{n - n_h}{n - n_{h-1}} \tag{18}$$

When the algorithm does return a tuple from Level h, it is easy to verify that all tuples with designated query at Level h are returned with equal probability. Note that since $k = 1$, the number of tuples with designated query at Level h is $n_h - n_{h-1}$. Thus, once LEVEL-SAMPLER reaches Level h, the probability for each tuple at Level h to be returned is $(1 - p(h))/(n_h - n_{h-1}) = 1/(n - n_{h-1})$.

Since LEVEL-SAMPLER reaches Level h iff it cannot retrieve the sample from Levels 0 to $h - 1$, for a given tuple with designated query at Level h, the probability for LEVEL-SAMPLER to select the tuple as a sample is

$$\left(\prod_{i=1}^{h-1} \left(1 - \frac{x_i}{n - n_{i-1}}\right)\right) \cdot \frac{1}{n - n_{h-1}} = \frac{1}{n} \tag{19}$$

The derivation is due to the fact that $n - n_i = n - n_{i-1} - x_i$. Thus, the algorithm generates unskewed simple random samples from tuples with designated queries at Levels 1 to C. □

4.1.3 *Analysis of Improvement on Efficiency*

Let $\pi^{-1}(i)$ be the maximum value of x that satisfies $pi(x) \leq i$. The following theorem shows that LEVEL-SAMPLER is significantly more efficient than HIDDEN-DB-SAMPLER and OVERFLOW-SAMPLER on sampling tuples with designated queries on or above Level C.

THEOREM 4.2. *The expected number of queries issued by LEVEL-SAMPLER to sample each Level-C-retrievable tuple with probability of* $1/n$ *is*

$$E(\textit{Cost of LEVEL-SAMPLER}) \leq \sum_{i=0}^{h} \frac{2^{i+1}}{n} \leq 4 \tag{20}$$

Such expected number for HIDDEN-DB-SAMPLER and OVERFLOW-SAMPLER is at least $\pi^{-1}(n/k) - 1$.

We omit the proof due to space limitation. Note that the upper bound derived for LEVEL-SAMPLER holds on arbitrary distribution of tuples.

4.2 Algorithm TURBO-DB-STATIC

Algorithm 5 depicts TURBO-DB-STATIC, an integration of all three ideas - leveraging overflowing queries, concatenating sampling with sampling, and level-by=level sampling. In the algorithm,

$$\rho(j, i) = k \cdot \prod_{b=j}^{i-1} |D_b| \tag{21}$$

where $|D_b|$ is the domain size of Attribute a_b. For a Boolean database, $\rho(j, i) = k \cdot 2^{i-j}$. Recall from Section 2 that $a_0, \ldots, a_{m-1}$ are the attributes. $card(q)$ is the number of tuples returned by q. Root is Level 0. n_0 is the pre-determined threshold for crawling. One can see that the algorithm utilizes the two efficiency-improving strategies, *query-tree pruning* and *breath-first sampling*:
Query-Tree Pruning: The simplest way to obtain the answer to a query is to issue the query (through the web interface), as is done in Algorithm 5. TURBO-DB-SAMPLER improves sampling efficiency by sending a query to the hidden database only if the query cannot be inferred from the historic query answers. For example, if a historic query returns valid or underflow, then no successor of the query needs to be issued, because its answer can be readily inferred from the historic query answer (by matching the returned tuples with the new query's search conditions). This strategy can be considered as pruning the query tree based on historic queries - the subtree of any issued underflowing and valid queries can be removed from the tree because the corresponding queries need not to be issued in the future.
Breath-First Sampling: TURBO-DB-SAMPLER also improves efficiency by performing breath-first search for the s samples to be collected i.e., for each given level h, execute LEVEL-SAMPLER for all of the s samples which have not been returned from higher levels. The premise of this idea is that one would prefer issuing higher-level queries first, in order to prune the search space for lower-level queries and to increase the cutoff level C.

Algorithm 5 TURBO-DB-STATIC

```
1: Set $h = 0$. $\Lambda(i) = \rho(0, i)$ for $i \in [0, m-1]$. $W[i] = \varnothing$ for $i \in [1, s]$.
2: for all $i \in [1, s]$ with $W[i] = \varnothing$ do
3:     With probability of $1 - \Lambda(h)/n$, Goto 16 for the next loop.
4:     Randomly choose a query $q$ from Level $h$. With probability of $card(q)/k$, randomly pick a tuple from the result of $q$ as $t$.
5:     if a valid or underflowing ancestor of $q$ was issued then
6:         Goto 4 if $t = \varnothing$.
7:     else if $q$ is valid or underflowing then
8:         for $i = h+1$ to $m$ do $\Lambda(i) = \Lambda(i) - \rho(h, i) + card(q)$
9:     end if
10:    if $t \neq \varnothing$ then
11:        Issue query $P(q)$, the parent of $q$.
12:        if $t \in P(q)$ then with probability of $\Lambda(h)/n$, Goto 4.
13:        else Set $W[i] = t$.
14:        end if
15:    end if
16: end for
17: if $\Lambda(h+1) \leq n$ then
18:     Set $h = h + 1$, Goto 2
19: end if //below switch to CONCATENATE-SAMPLER
20: Randomly select an overflowing query $q$ from Level $h$.
21: Crawl the subtree of $q$ with $n_{\mathrm{T}}$ tuples below Level $h$.
22: With probability of $n_{\mathrm{T}}/n_0$, return a tuple chosen uniformly at random from the $n_{\mathrm{T}}$ tuples, otherwise Goto 20.
```

5. DISCUSSIONS

5.1 Uniform Random Sampling vs Weighted and Stratified Sampling

As we mentioned in the introduction, traditional database sampling can be done in a variety of ways besides uniform random sampling. Two popular techniques are stratified sampling and weighted sampling. In this subsection, we discuss our choice of uniform random sampling for hidden databases.

A unique challenge to hidden database sampling is the lack of access to the population being sampled. This presents a significant obstacle to stratified sampling, because it has to (1) select an appropriate stratification variable which partitions all tuples into relatively homogeneous subgroups, and (2) determine the size of each subgroup. While these two tasks can be easily done for a traditional database [10], it is not immediately clear how they can be accomplished for a hidden database without incurring significant query cost.

Another possible way to sampling hidden database is weighted sampling. Such a scheme has been proposed for sampling search engines [3] to estimate aggregate query answers. Adapted to hidden database sampling, it works as follows: instead of performing rejection sampling on a tuple before selecting it as a sample, this scheme always selects such a tuple while associating it with a *weight* computed from the probability that such a tuple is selected. For example, consider a hidden database with $k = 1000$ and two attributes a_1, a_2. There are 1000 tuples with $a_1 = 0$ and 1 with $a_1 = 1$. If we adapt HIDDEN-DB-SAMPLER to the weighted sampling scheme, the tuples with $a_1 = 0$ will be assigned a weight of $w_0 = 2000/1001$ while the tuple with $a_1 = 1$ has weight of $w_1 = 2/1001$. This way, $\forall i \in \{0, 1\}$,

$$\Pr\{\text{a tuple with } a = i \text{ is selected}\} \cdot w_0 = \frac{1}{1001}. \quad (22)$$

A main problem of this weighted sampling scheme is that its accuracy on answering an aggregate query depends on whether the selection probability distribution of all tuples is *aligned* with the aggregate query. To understand why, consider the above example and an aggregate query `SELECT AVG(`a_2`) FROM` D. To estimate the answer to this query, the weighted sampling scheme computes the mean of $t[a_2] \cdot w(t)$ for all sample tuples, where $t[a_2]$ and $w(t)$ are the value of a_2 and the weight for a sample tuple t, respectively. If $a_2 = 1.001$ and 1001 for all tuples with $a_1 = 0$ and 1, respectively, then the weighted sampling performs perfectly with zero standard error because $1.001 \cdot w_0 = 1001 \cdot w_1 = 2$.

However, if $a_2 = 1001$ and 1.001 for tuples with $a_1 = 0$ and 1, respectively, the weighted sampling scheme performs significantly worse than uniform random sampling. Note that in this case, $t[a_2] \cdot w(t) = 2000$ and 0.002, respectively. The variance of it is $((2000 - 1000.001)^2 + (0.002 - 1000.001)^2)/2 = 999998$. In comparison, the variance of $t[a_2]$ for a sample tuple t generated by uniform random sampling is $(1000/1001) \cdot (1001 - 1000.001)^2 + (1/1001) \cdot (1.001 - 1000.001)^2 = 998.001$. Note that when issuing the same number of queries, the weighted sampling scheme generates $1/(1/2 + 1/2000) = 1.998$ times as many samples as the original HIDDEN-DB-SAMPLER. Thus, given the same query cost, the standard error produced by weighted sampling is $\sqrt{999998/(998.001 \cdot 1.998)} = 22.394$ times that of uniform random sampling.

Since we do not assume knowledge of the aggregate query to be estimated in this paper, we choose uniform random sampling as our objective. We shall investigate the design of sampling techniques with knowledge of the target aggregate queries in the future work.

5.2 Extensions for Numerical Databases

For numerical databases, a sampler can first discretize the numerical data to resemble categorical data before applying the sampling algorithms discussed in this paper. However, different discretization techniques have different impact on the performance of sampling. Given our preference of minimizing n_0 to reduce the quest cost, a equal-size discretization might be preferred. Nonetheless, how to determine the optimal number of intervals and choose an optimal discretization scheme is left as an open problem.

6. EXPERIMENTAL RESULTS

In this section, we describe our experimental setup and compare our TURBO-DB-SAMPLER and TURBO-DB-STATIC with the exiting HIDDEN-DB-SAMPLER [12] and HYBRID-SAMPLER algorithms [13].

6.1 Experimental Setup

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

Yahoo! Auto Dataset: The Yahoo! Auto dataset consists of data crawled from a real-world hidden database at *http://autos.yahoo.com/*. In particular, it contains 200,000 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.

System Implementation: We set $k = 100$. Since the database query processing technique is relatively deterministic, we implemented our own in-memory database engine using Matlab. All tuples are ranked lexicographically.

Parameter Settings: The experiments involve four algorithms: HIDDEN-DB-SAMPLER [12], HYBRID-SAMPLER [13], as well as TURBO-DB-SAMPLER and TURBO-DB-STATIC introduced in this paper. TURBO-DB-SAMPLER and TURBO-DB-STATIC require one parameter: the cutoff level C for switching from sampling to crawling. We conducted experiments with various values of C ranging from 5 to 15. HIDDEN-DB-SAMPLER also requires the cut-off level C as a parameter to balance between efficiency and skew. Following the heuristic in [12], we set C for HIDDEN-DB-SAMPLER to be the average length of random walks that reach a valid query. HYBRID-SAMPLER requires two parameters: the number of pilot samples s_1 and the switching count threshold c_S. We set $s_1 = 20$ and $c_S = 5$.

Performance Measures: For each algorithm, there are two performance measures: *efficiency* and *skew*. Efficiency of a sampling algorithm was measured by counting the number of unique queries that were executed to reach a certain desired sample size. For measuring the skew of collected samples, there has not been a widely accepted measure. Thus, we follow the same measure as [12] which compares the marginal frequencies of attribute values in the original dataset and in the sample:

$$skew = \sqrt{\frac{\sum_{v \in V} \left(1 - \frac{p_S(v)}{p_D(v)}\right)^2}{|V|}}. \qquad (23)$$

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). Again, even unskewed samples may have small but possibly non-zero value of the measure.

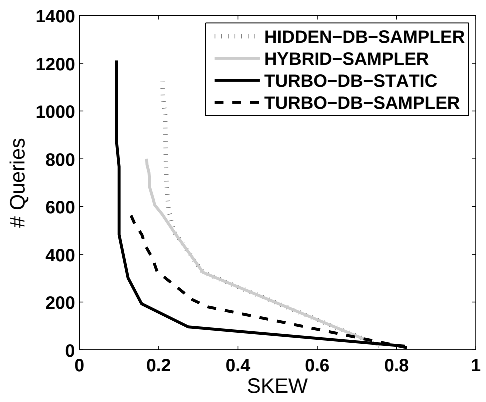

Figure 3: A sample line graph using colors which contrast well both on screen and on a black-and-white hardcopy

6.2 Comparison with State-of-The-Art Samplers

We tested TURBO-DB-SAMPLER and TURBO-DB-STATIC on the Yahoo! Auto dataset and compared them against HIDDEN-DB-SAMPLER and HYBRID-SAMPLER in terms of the tradeoff between efficiency and skew. Figure 3 depicts the results. For TURBO-DB-SAMPLER, we set C to be the average length from the root to the designated query of a tuple. For TURBO-DB-STATIC, we set C to be the maximum value that satisfies the requirement of $k \cdot \pi(C) \leq n$. The different points in the figure are generated by varying the number of sample tuples to be collected between 1 and 500. One can see from the figure that the samplers we proposed in this paper provides significant improvement over the existing algorithms on the tradeoff between efficiency and skew.

Figure 4 and Figure 5 depicts the change of query cost and skew during the process of collecting 100 samples, respectively. Note that only TURBO-DB-SAMPLER, HIDDEN-DB-SAMPLER, and HYBRID-SAMPLER are shown in these figures due to the breath-first sampling scheme we used for TURBO-DB-STATIC - i.e., it collects all samples for one level before going to the next. One can see from the figures that TURBO-DB-SAMPLER significantly outperforms the existing algorithms in terms of both efficiency and skew.

6.3 Evaluation of Parameter Settings

We now investigate the change of efficiency and skew of our algorithms with the cut-off level parameter C. Figure 6 depicts the change of query cost and skew for TURBO-DB-SAMPLER to collect 100 samples when C varies between 7 and 15. One can see that while the skew is reduced when C increases, the number of queries remain roughly constant for different values of C. This is because with a larger C, while the OVERFLOW-SAMPLER part of TURBO-DB-SAMPLER needs to issue more queries due to the reduced acceptance probability, the CONCATENATE-SAMPLER part, however, requires fewer queries because the subtree that needs to be crawled becomes smaller.

Figure 7 depicts the change of query cost and skew for TURBO-DB-STATIC to collect 100 samples when C varies between 5 and 10 (the maximum value allowed by the algorithm requirement of $k \cdot \pi(C) \leq n$). One can see that unlike TURBO-DB-SAMPLER, a larger value of C reduces both skew and query cost for TURBO-DB-STATIC. This is because when C is too small (e.g., 5 or 6 in our dataset), the query cost is dominated by crawling, the cost of which can be significantly reduced with a larger C. The value assignment of C we suggested in the paper is 10 for this dataset which, as one can see from the figure, achieves a fairly good tradeoff between efficiency and skew.

7. 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. [1, 19, 21] deal with extracting data from structured hidden databases. [8] and [20] use query based sampling methods to generate content summaries with relative and absolute frequencies while [17,18] uses two phase sampling method on text based interfaces. On a related front [7, 9] 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 [6] and more recently [2, 5]. In [12] and [13] the authors have developed techniques for random sampling from structured hidden databases leading to the HIDDEN-DB-SAMPLER, COUNT-BASED-SAMPLER, and HYBRID-SAMPLER algorithms.

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 [11] and Garofalakis et al [15], as well as [4] and the references therein.

8. CONCLUSION

In this paper, we investigated techniques which leverage overflowing queries to efficiently sample a hidden database. In particular, we proposed TURBO-DB-SAMPLER which significantly improves sampling efficiency while allowing much smaller skew than state-of-the-art samplers. We also proposed TURBO-DB-STATIC, an algorithm that achieves additional speedup for databases with static scoring functions. Our thorough experimental study demon-

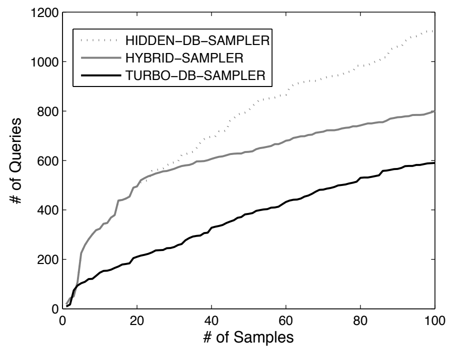

Figure 4: Number of Queries vs. Number of Samples

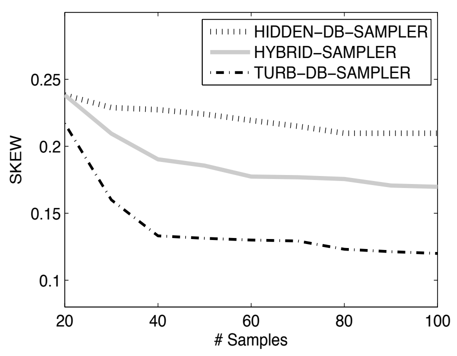

Figure 5: Level of Skew vs. Number of Samples

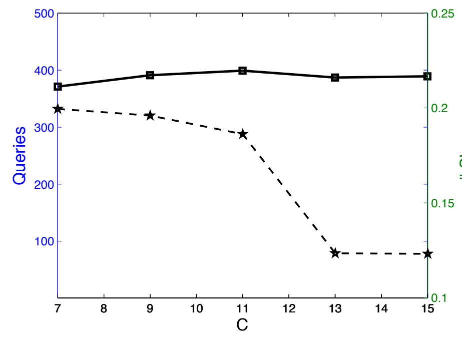

Figure 6: Change of C for TURBO-DB-SAMPLER

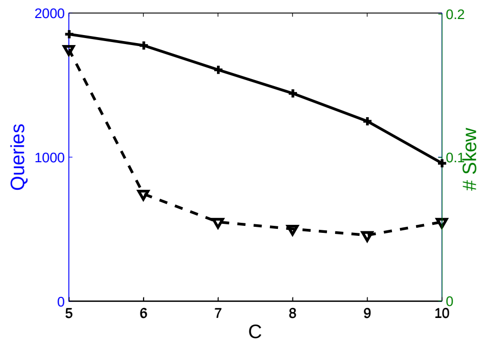

Figure 7: Change of C for TURBO-DB-STATIC

strates the superiority of our sampling algorithms over the existing algorithms.

9. REFERENCES

[1] M. Alvarez, J. Raposo, A. Pan, F. Cacheda, F. Bellas, and V. Carneiro. Crawling the content hidden behind web forms. In *ICCSA*, 2007.

[2] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. In *WWW*, 2006.

[3] Z. Bar-Yossef and M. Gurevich. Efficient search engine measurements. In *WWW*, 2007.

[4] 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*, 20(4):3–45, 1997.

[5] L. Barbosa and J. Freire. Siphoning hidden-web data through keyword-based interfaces. In *SBBD*, 2004.

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

[7] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[8] J. P. Callan and M. E. Connell. Query-based sampling of text databases. *ACM TOIS*, 19(2):97–130, 2001.

[9] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[10] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 32(2), 2007.

[11] G. Das. Survey of approximate query processing techniques (tutorial). In *SSDBM*, 2003.

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

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

[14] A. Dasgupta, N. Zhang, G. Das, and S. Chaudhuri. Privacy preservation of aggregates in hidden databases: Why and how? In *SIGMOD*, 2009.

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

[16] Google Base. *http://base.google.com*.

[17] 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.

[18] Y.-L. Hedley, M. Younas, A. E. James, and M. Sanderson. Sampling, information extraction and summarisation of hidden web databases. *Data and Knowledge Engineering*, 59(2):213–230, 2006.

[19] S. W. Liddle, D. W. Embley, D. T. Scott, and S. H. Yau. Extracting data behind web forms. In *ER (Workshops)*, 2002.

[20] L. G. Panagiotis G. Ipeirotis. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB*, 2002.

[21] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, 2001.

[22] Yahoo! Auto. *http://auto.yahoo.com*.