Rank Discovery From Web Databases *

Saravanan Thirumuruganathan[†,‡], Nan Zhang[††], Gautam Das[†,‡]
[†]University of Texas at Arlington; [††]George Washington University

ABSTRACT

Many web databases are only accessible through a proprietary search interface which allows users to form a query by entering the desired values for a few attributes. After receiving a query, the system returns the top-k matching tuples according to a pre-determined ranking function. Since the rank of a tuple largely determines the attention it receives from website users, ranking information for any tuple - not just the top-ranked ones - is often of significant interest to third parties such as sellers, customers, market researchers and investors. In this paper, we define a novel problem of rank discovery over hidden web databases. We introduce a taxonomy of ranking functions, and show that different types of ranking functions require fundamentally different approaches for rank discovery. Our technical contributions include principled and efficient randomized algorithms for estimating the rank of a given tuple, as well as negative results which demonstrate the inefficiency of any deterministic algorithm. We show extensive experimental results over real-world databases, including an online experiment at Amazon.com, which illustrates the effectiveness of our proposed techniques.

1. INTRODUCTION

1.1 The Rank Discovery Problem

Many web databases, e.g., Yahoo! Autos, Amazon.com, are "hidden" behind (i.e., only accessible via) a restrictive form-like interface which allows a user to form a search query by specifying the desired values for a few attributes; and the system responds by returning a small number of tuples matching the search query. Almost all such interfaces enforce the top-k constraint - i.e., when more than k tuples (where k is typically a predetermined small constant) match the user-specified query, only k of them are preferentially selected according to a ranking function and returned to the user. While such restrictive form interfaces of hidden databases might suffice for the simplest use-cases, i.e., that of a normal user searching for some items in these databases, they often cannot satisfy users with specific needs and also prevent many interesting third-party services from being developed over web databases. There has been several recent works on developing techniques to enable additional functionality over such databases that operate via the restrictive interface, such as sampling and aggregate estimation (see [8, 20, 1] and references therein).

In this paper, we consider a novel problem, that of discovering rank-related information from a hidden web database:

> RANK DISCOVERY PROBLEM: Given a query q, and a tuple t that satisfies the selection conditions of q, estimate $r(t, q)$, the rank of t among all tuples in the web database that satisfy q, as accurately as possible with minimal query cost.

In a general sense, the rank discovery problem is a fundamental data analytics task, because it seeks to determine the rank/position of an item as compared to similar competing items along multiple attributes/facets. In the case of web databases, since the rank of a tuple largely determines the attention it receives from website users, ranking information for any tuple - not just the top-k ranked ones - is often of significant interest to third parties such as sellers, customers, market researchers and investors. Solutions to the rank discovery problem could enable new third-party application scenarios that have not been considered in earlier work. For example, the author of a book on sale at Amazon would like to monitor the ranking of her book within a set of similar competitors (e.g., how does it rank in sales, or customer reviews, etc., compared to other similar books on science fiction?). Likewise, competitors to an app available at Apple's iOS and Mac App Stores would be interested in monitoring the app's grossing rank and measure the market response to determine if it is time to start competing with the app.

Although the rank discovery problem appears deceptively simple, it is challenging because most web databases do not explicitly disclose a tuple's rank beyond the top-k tuples. The rank has to be discovered indirectly, by carefully issuing multiple related queries to the website's query interface and inferring the tuple rank by piecing together the information returned from these queries. We observed that different websites have widely varying characteristics, resulting in a myriad of interesting facets and variants of the rank discovery problem that require fundamentally different approaches in their solutions. In the rest of this introductory section, we provide an overview of this spectrum of problem variants, highlight

*The work of Saravanan Thirumuruganathan and Gautam Das is partially supported by NSF grants 0812601, 0915834, 1018865, a NHARP grant from the Texas Higher Education Coordinating Board, and grants from Microsoft Research and Nokia Research. The work of Nan Zhang is partially supported by NSF grants 0852674, 0915834, and 1117297. Any opinions, findings, conclusions, and/or recommendations expressed in this material, either expressed or implied, are those of the authors and do not necessarily reflect the views of the sponsor listed above.
‡ Part of work done while at Qatar Computing Research Institute.

Proceedings of the VLDB Endowment, Vol. 6, No. 13

their difficulties and challenges, and summarize our technical contributions - both algorithmic as well as negative results.

1.2 Problem Variants and Challenges

Web databases use a broad variety of ranking functions. These functions typically compute a score for each tuple matching the query conditions, and return the k tuples with the highest scores. These ranking functions can be classified along several different dimensions. One dimension is whether the function is *static* or *query dependent*. A static ranking function assigns tuple scores independent of the query - i.e., all tuples are globally ordered in the database. For example, Amazon allows users to search for books by specifying a few desired attributes (e.g., Language, Format, Genre, Release date, Title, etc.), and the system returns up to k matching books, ranked by price, average customer review, popularity (i.e., sales amount), recency, etc. One can see that all these ranking functions are static. Other examples are the "sort by bestsellers" or "sort by grossing" static ranking functions used by Apple's iOS and Mac App Stores. A ranking function is query-dependent if the score of a tuple varies for different queries - e.g., where all tuples are ordered according to the number of attribute matches between the query and each tuple, or by a more sophisticated notion of "relevance".

Within static ranking functions, a second dimension for categorization is whether the function is *observable* or *proprietary* (i.e., unobservable). Observable ranking functions are those where the end-user can determine the score of a tuple from the tuple values alone - e.g., in the aforementioned Amazon example, if the ordering is by price, the score of a tuple is obvious. Observable ranking functions may be further categorized into whether the scoring attribute can be *queried* or not. For example, users can query for products in Amazon by specifying desired price ranges, but cannot specify desired recency, although both scores are observable in returned products. A proprietary ranking function is one where the tuple's score is hidden from the public's view - e.g., the actual values of popularity (i.e., sales amount) and/or gross sales for Amazon and App Stores are never revealed to the end user.

We note that no matter what variant of web database we encounter, trivial solutions to the rank discovery problem are possible if (1) all input tuples are very highly ranked so as to enter the top-k results returned by the hidden database, and/or (2) the third-party analyzer can negotiate a private agreement with the web database owner in order to retrieve the ranking of the entire query results beyond the top-k limitation. Nonetheless, note that the information most useful to an investor and/or competitor occurs *before* a book/app enters the top-seller list and becomes more or less known to the general public anyway. On the other hand, private negotiations are often very difficult due to revenue sharing, legal requirements, security and myriad of other thorny issues. As such, our focus in this paper is to develop automated third-party algorithms that only use the public interfaces of web databases without requiring any additional cooperation from the database owners.

Another seemingly straightforward solution to issue all possible queries through the web interface so as to crawl all rank-related information one could possibly infer from the hidden database - and then analyze the query answers locally to address the above problems. Nonetheless, a key pitfall of this solution is its prohibitively high query cost [17] - which is simply infeasible for real-world web databases which often impose a per user/IP limit on the number of queries one can issue over a given time frame (e.g., Google Search API allows only 100 free queries per user per day).

Given the pitfalls of the above mentioned approaches, in this paper we develop algorithms to solve the rank discovery problem only using the public interfaces of web databases, and with minimal *small query cost* - a goal shared by most existing studies on exploring hidden web databases because of the query-number limitations enforced by web databases (e.g., [8]). Our algorithms produce approximate answers, and thus another important design objective is to produce answers with small relative error.

1.3 Outline of Technical Results

As mentioned above, different types of ranking functions require fundamentally different approaches for rank discovery. For ranking functions that are static, observable, and can be queried through the search interface, we show that a simple solution exists for rank discovery: use the existing aggregate estimation algorithms [8] to estimate the COUNT of tuples with a higher rank. If the ranking function is query dependent, then there is obviously no way to determine the rank of a tuple outside the top-k ones returned by the query - because no other query reveals its query-dependent rank information. For the remaining cases - i.e., when the ranking function is (i) static and proprietary, or (ii) static, observable but cannot be queried - we develop RANK-EST, a rank estimation algorithm that interleaves the following two methods:

- RANK-EST-S, a sampling-based process which first draws uniform random samples from the hidden databases, and then perform rank companions between the input tuple and the samples to enable rank estimation.
- RANK-EST-H, a randomized process which randomly constructs and issues queries that return tuples ranked higher than the input tuple, and use the query answers to directly produce a rank estimation.

While RANK-EST-S works well for most tuples in the database, it cannot effectively handle highly ranked tuples, because a very large sample is required to accurately estimate their ranks. RANK-EST-H, on the other hand, is designed specifically for these tuples. As such, by interleaving the two processes, RANK-EST achieves efficient and accurate rank estimation over all tuples in the database.

Interestingly, while RANK-EST works for both proprietary and observable ranking functions, the technical challenges facing the design of RANK-EST-S (and consequently, the query cost required by it) differs drastically between the two. Specifically, while comparing rank between the input and a sample tuple is straightforward for observable ranking functions, it can be extremely difficult for proprietary ones. We prove a hardness result showing that no deterministic algorithm can do so without issuing an extremely large number of queries. To address this problem, we devise LV-RANK-COMPARE, a randomized rank comparison algorithm, inside RANK-EST-S. LV-RANK-COMPARE is a Las-Vegas algorithm - i.e., it always produces the correct answer, but with varying query costs across different executions.

1.4 Summary of Contributions

- We introduce and motivate the novel problem of rank discovery over hidden web databases.
- We define a comprehensive spectrum of ranking functions according to various dimensions such as query-dependent vs. static, observable vs. proprietary, and whether the scoring attribute can be queried or not. We discuss the feasibility of rank discovery for each type of ranking function, and show that different types of ranking functions require fundamentally different approaches for rank discovery.
- For proprietary and observable ranking functions, we develop RANK-EST which interleaves two separate procedures for handling high and low ranked tuples, respectively.
- We present careful theoretical analysis including negative results that preclude efficient deterministic solutions.

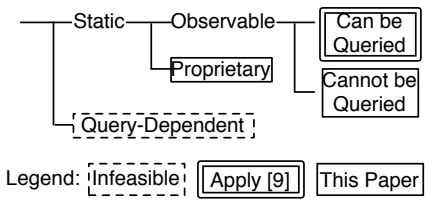

Figure 1: Taxonomy

- We present a thorough experimental evaluation of our algorithms over real-world hidden web databases. We also describe online experiments over Amazon.com which demonstrates the effectiveness of our proposed algorithms.

2. RANK DISCOVERY PROBLEM

In this section, we introduce a taxonomy of ranking functions commonly used and, for each type of ranking functions, discuss the feasibility of rank discovery. Then, we define the technical problem addressed in the paper.

2.1 Model of Hidden Databases

Consider a hidden database D with n tuples and m input attributes $A_1, A_2, \ldots, A_m$. Given a tuple t and an attribute A_i, let $t[A_i]$ be the value of A_i in t. Let $Dom(A_i)$ be the domain of A_i. For the purpose of this paper, we restrict our attention to categorical attributes and assume the appropriate discretization of numeric ones. We also consider all tuples distinct and without null values.

We consider a user query q to be a collection of *conjunctive* constraints of the form `SELECT * FROM` D `WHERE` $A_{i_1} = v_{i_1}$ $\& \ldots \& A_{i_s} = v_{i_s}$, where $i_1, \ldots, i_s \in [1, m]$ and $v_{i_j} \in Dom(A_{i_j})$. The hidden databases generally restrict users to top-k tuples- which may be presented on one page or over multiple pages (accessed by page turns or clicking next at the bottom of the results page)[1].

Formally, let the set of tuples matching q be $Sel(q)$. If $|Sel(q)| > k$, an $overflow$ occurs and only the top-k results are returned, along with an overflow flag indicating that more tuples matching the query cannot be returned. If $|Sel(q)| = 0$, then an *underflow* is said to occur. Otherwise, i.e., when $|Sel(q)| \in [1, k]$, we say that q is valid - i.e., the user retrieves all tuples matching the issued query. While our running example below uses a Boolean database for ease of exposition, all our results hold for any categorical database.

Running Example: Table 1 shows a simple table which we use as running example throughout this paper. There are m = 5 Boolean attributes and n = 8 tuples.The tuples are listed in the order of their rank - i.e., t_1 is top-ranked.

2.2 Taxonomy of Ranking Functions

Ranking function is what the hidden database uses to determine which k tuples to return when a query overflows. Consider a ranking function $f(\cdot)$ which takes a tuple and a query as input and outputs a score. There are two broad categories of ranking functions: *static* and *query-dependent*. A ranking function $f(\cdot)$ is *static* if $\forall q_1, q_2$, $f(q_1, t) = f(q_2, t)$. Otherwise, it is *query-dependent*.

Within static ranking functions, we can further partition them into two types, *observable* or *proprietary*. A ranking function is *observable* if $f(t)$ is displayed along with (or can be inferred from) other attributes when a tuple is returned in a query answer. Otherwise, it is proprietary. Examples of observable ranking functions include Price, Listed Date, etc., used by many e-commerce websites. Proprietary ranking functions include "Popularity" in numerous websites such as Amazon.com, Priceline, Kickstarter, etc., "sort by gross sales" used by Apple's and Android's App Stores, as well as many proprietary scoring functions such as Moviemeter in IMDB and "Rank by value" in Seatguru.

Finally, within observable (static) ranking functions, we can further partition them into two categories according to whether the scoring attribute can be *queried* through the search interface. Specifically, a ranking function can be queried if it is possible to issue `SELECT * FROM D WHERE` $f(t) = c$ through the interface. An example is the Recency ranking function used by Amazon.com. Figure 1 depicts the above taxonomy of ranking functions.

[1]For example, Google limits the number of page turns to 10 if 100 results are displayed per page, and 100 if 10 results per page - effectively resulting in a top-1000 interface.

TABLE 1. Running Example

	A_1	A_2	A_3	A_4	A_5
t_1	0	0	0	0	1
t_2	0	0	0	1	1
t_3	0	0	1	0	1
t_4	0	1	1	1	1
t_5	1	1	1	0	1
t_6	1	1	1	1	1
t_7	1	0	0	0	0
t_8	0	0	0	0	0

2.3 Feasibility of Rank Discovery

In this subsection, we consider the feasibility of rank discovery over four types of ranking functions: (i) query-dependent, (ii) static, observable can be queried through the interface, (iii) static, observable but cannot be queried, and (iv) static and proprietary, respectively. Figure 1 summarizes the following feasibility results:

Query-Dependent Ranking Functions: For a query-dependent ranking function, there exists no *fixed* ordering among tuples (as changing the query, changes the order of tuples) which makes the rank discovery problem meaningless. Hence, unless the input tuple is in the top-k of the query, no mechanism can compute its rank.

Observable and Queriable Ranking Functions: At the other extreme - when a ranking function is static, observable, and can be queried through the search interface - rank discovery can be reduced to the problem of aggregate query processing, which has been addressed in existing work [8]. The reason is simple - since `SELECT * FROM D WHERE` $f(t) = c$ is supported by the interface, the aggregate estimation algorithms in [8, 20, 1] can be readily applied to estimate `SELECT * FROM D WHERE` $f(t) > f(t_0)$, which is exactly the rank of input tuple t_0.

Observable and Non-Queriable Ranking Functions: When an observable ranking function is nevertheless not queriable through the interface, the simple solution described above no longer applies. However, rank discovery is always feasible - for example, a naive approach is to crawl all tuples from the database, and then rank them locally according to observations of the scoring attribute.

Proprietary Ranking Functions: For proprietary ranking functions, a key concept for understanding the feasibility of rank discovery is the *direct domination* relationship between two tuples. To understand the concept, consider what rank-related information a query answer q reveals. For the (at most) k tuples returned by q, their ranks can be compared according to the query answer. In addition, it is also possible to infer from q rank-related information for tuples that are *not* returned by it. Specifically, for two tuples t and t' which *match* q, we can determine which has a higher rank if (at least) one of them is returned by q:

- if q returns t but not t', then t is ranked higher,
- if q returns t' but not t, then t' is ranked higher, or
- if q returns both, then we can make the comparison based on the returned order.

For two given tuples, if there exists a query such that any of the three cases occurs, we say the two tuples are *directly comparable* with each other, with the higher-ranked tuple *directly dominating* the other one.

DEFINITION 1. **[Direct Domination]** *A tuple t is said to **directly dominate** another tuple t', i.e., $t \succ t'$, if and only if t and t' are directly comparable and t ranks higher than t'.*

Consider the running example shown in Table 1 with $k = 2$. We can observe that t_1 and t_3 are directly comparable using the query q_1: `SELECT * FROM D WHERE` $A_1 = 0$ `AND` $A_2 = 0$ `AND` $A_4 = 0$ `AND` $A_5 = 1$ with t_1 ranked higher than t_3. Similarly, we can see that tuple t_2 directly dominates t_3 using the query q_2: `SELECT * FROM D WHERE` $A_1 = 0$ `AND` $A_2 = 0$ `AND` $A_5 = 1$. The result includes t_2 but not t_3 - i.e., t_2 ranks higher.

Given the direct domination relationships, the feasibility of rank discovery for a tuple t boils down to whether, for all other tuples t' in the database, it is possible to find a sequence of tuples $t_1, \ldots, t_h$, such that $t \succ t_1 \succ \cdots \succ t_h \succ t'$ or vice versa. If the chain can be found for all other tuples, then the rank of t can be precisely discovered. Otherwise, the fewer chains we can find, the wider a range we have to settle on estimating the rank. We call this problem the potential *discrepancy between real and revealed ranks*.

Fortunately, as we shall show in §6.2 and §7, while the discrepancy problem does exist in theory, the probability for it to occur in practice is extremely low - i.e., in almost all cases, the real rank is exactly disclosed by the top-k interface. Thus, rank discovery is indeed feasible for proprietary ranking functions.

Computing $\mathcal{D}(t)$: Given a tuple t, one can obtain the set of all tuples that directly dominate t, denoted by $\mathcal{D}(t)$, by issuing all queries that match t - i.e. those that have predicates in the power-set of $\{t[A_1], t[A_2], \ldots, t[A_m]\}$. Note that the number of queries that have j predicates matching t is $\binom{m}{j}$. Thus, one needs to issue a total of $\binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m} = 2^m$ queries to compute $\mathcal{D}(t)$. This result holds for both Boolean and categorical databases.

Before concluding this subsection, we make an important observation that, if two tuples are *directly* comparable, then we need only *one* query to determine their domination relationship: the *most specific* query which matches both tuples - i.e., the query which contains one predicate for each attribute on which both tuples share the same value. If this query cannot return at least one of the two tuples, then no other query can - i.e., the two tuples are not directly comparable. In the running example, q_1 is the most specific query matching t_1 and t_3, while q_2 is the one matching t_2 and t_3.

2.4 Technical Problem

Discussions in the above subsection leave us with two types of ranking functions for which the rank discovery problem is feasible and unsolved - those that are proprietary, and those that are observable but cannot be queried. As discussed in §1, both types are widely prevalent in practice. Thus, we focus on solving the rank discovery problem for these two types in the paper.

Objective of Rank Discovery: Intuitively, the objective of rank discovery is to find the rank of a given tuple, i.e., the number of tuples with a higher rank than the given tuple, within a user-defined subset of the hidden database - which we model using a filtering query q_{F}. For example, a user may be interested in the rank of a car within all Honda Accords, in which case q_{F} is `SELECT * FROM D WHERE` Make = Honda `AND` Model = Accord. If a user is interested in the global rank within the entire database, then q_{F} is `SELECT * FROM D`. In this paper, we support any filtering query q_{F} as long as whether $t \in Sel(q_{\text{F}})$ can be determined solely upon knowledge of q_{F} and t (and not other tuples in the database[2]).

Given the closeness of real and revealed ranks as discussed in §2.3, we define the problem of rank discovery as the extraction of a tuple's rank revealed through the top-k interface. Specifically,

DEFINITION 2. **[Problem Definition]** *Given a hidden database D and a filtering condition q_{F}, The objective of rank discovery is to compute $r(t, q_{\text{F}}) = |\Omega(t, q_{\text{F}})|$, where $\Omega(t, q_{\text{F}}) \subseteq Sel(q_{\text{F}})$ satisfies that $\forall t' \in \Omega(t, q_{\text{F}})$, either $t' \succ t$ or there exists tuples $t_1, \ldots, t_h \in D$, such that $t' \succ t_1 \succ \cdots \succ t_h \succ t$, where $\succ$ is the direct domination relationship defined in Definition 1.*

Performance Measures: The performance is measured using *query cost* - the number of queries one has to issue through the web search interface of the hidden database. Accuracy-wise, we consider the *relative error* measure for an estimated rank $\tilde{r}(t, q_{\text{F}})$ as

$$\delta = \frac{|r(t, q_{\text{F}}) - \tilde{r}(t, q_{\text{F}})|}{r(t, q_{\text{F}})}. \quad (1)$$

Note that, compared with the absolute error measure (i.e., $|r(t, q_{\text{F}}) - \tilde{r}(t, q_{\text{F}})|$), relative error is more meaningful in practice. To see why, consider an example where an algorithm produces $\tilde{r}(t_1, q_{\text{F}})$ = 200 for a 100-th ranked tuple and $\tilde{r}(t_2, q_{\text{F}})$ = 98761 for a tuple with rank 98661. While the error on t_1 represents a significant misconception of t_1's status in the database, the error on t_2 is hardly noticeable - yet both have the same absolute error of 100. Hence, we focus on the relative error measure in our theoretical analysis.

3. OVERVIEW OF TECHNICAL APPROACH

In this section, we start by describing three baseline techniques and their respective problems. Then, we provide an overview of our technical approach to address these problems - with details discussed in §4 and 5. Note that, in this section and for most part of the paper, we focus on discovering the *global rank* of a tuple within the entire database (i.e., when q_{F} in Definition 2 is `SELECT * FROM D`). Then, we shall discuss in §6.1 a simple extension to support other q_{F}. To simplify the notations, we denote the global rank of t by $r(t) = r(t, \texttt{SELECT * FROM D})$.

3.1 Baseline Techniques

We start by describing three baseline ideas for solving the rank discovery problem, and point out their respective problems which motivate our proposed design of RANK-EST in this paper.

Crawling: The first baseline idea is to *crawl* all tuples and the associated rank information from a hidden database, and then rank all tuples locally (as in traditional databases) to derive the rank of the input tuple. The main problem of this approach is the extremely high query cost incurred by crawling, especially for large hidden databases. Lower-bound results derived in [17] show that crawling requires a prohibitively high cost of at least $\Omega(m \cdot n^2/k^2)$ queries for certain categorical databases with a top-k interface - where m and n are the number of attributes and tuples, respectively.

Sampling: The second baseline approach is to first draw a uniform random *sample* of the hidden database [7, 9, 10], and then compare the rank of the input tuple with all sample tuples to extrapolate its rank in the database. This approach has two main problems:

First, it is unclear how to effectively compare the ranks of two given tuples when the ranking function is proprietary. Note that while doing so for two directly comparable tuples (as defined in

[2]e.g., queries such as `SELECT * FROM D WHERE` Price > `(SELECT AVG(`Price`) FROM D)` are not supported.

Definition 1) are easy, many pairs of tuples cannot be directly compared. For example, t_6 and t_8 in the running example do not have a direct domination relationship, because the only query which matches both tuples is SELECT * FROM D which returns neither. But one can still compare t_6 with t_8 by using t_7 as a "bridge", because t_6 is returned by $A_1 = 1$, while both t_7 and t_8 are returned by $A_5 = 0$. How to find such bridges, however, is a challenge for applying the sampling idea to proprietary ranking functions.

The second problem with this sampling-based approach is that the estimated rank may not be accurate enough for *highly ranked* tuples, unless one incurs a very high query cost to draw a large sample. To understand why, note that according to the accuracy measure in (1), to achieve the same accuracy level, the absolute error of rank estimation has to be much smaller for highly ranked tuples than lower ranked ones. On the other hand, it is easy to see that the sample size is inversely proportional to the square of absolute error - i.e., one needs a much larger sample for highly ranked tuples. For example, for a 1-million tuple database with $k = 50$, just to estimate a 100-th ranked tuple's rank within a relative error of 50%, the sampling-based approach needs at least an expected number of 1,000,000 / 100 = 10,000 samples - which could mean hundreds of thousands of queries even with the state-of-the-art sampler [10].

Ordered Crawling: The third baseline we consider is *ordered crawling*. The key idea here is to crawl tuples in the descending order of their ranks. GETNEXT primitive described in [19] retrieves the No. h ranked tuple given the top-(h-1) tuples as input. While [19] solves an entirely different problem of retrieving top-h tuples through a top-k interface ($h > k$), one can see that GETNEXT is capable of obtaining the rank of a highly ranked tuple without incurring the query cost for a complete crawl of the database. However, not only is it unsuitable for lowly ranked tuples[3], even for highly ranked tuples the query cost can be very high. For example, when $k = 100$, GETNEXT requires a prohibitive query cost of nearly 60,000 queries for a 2,000-th ranked tuple in a 200,000-tuple database [19]. Please see §8 for addition discussion.

3.2 Overview of Our Approach

Technical Challenges: Given the pitfalls of these three baseline approaches, we identify two key technical challenges for supporting efficient rank estimation:

- *Effective Rank Comparison:* To address the problem of applying sampling-based estimation to proprietary ranking functions, a key challenge is to efficiently compare the ranks of two tuples that do not have a direct domination relationship.
- *Efficient Rank Estimation for Highly Ranked Tuples:* Since the sampling-based approach is not effective for highly ranked tuples, a further challenge is to efficiently position a highly ranked tuple without crawling all higher-ranked ones.

Roadmap of Our Approach: In the remaining part of the paper, we shall address the two challenges respectively, before combining the techniques to form a comprehensive solution to rank discovery.

Specifically, we start by addressing the rank comparison problem for proprietary ranking functions in §4. We first describe a deterministic algorithm and its problem with high query cost - which motivates us to propose a efficient probabilistic solution. An interesting feature of this probabilistic solution is that it is a *Las Vegas algorithm* - i.e., it always produces the correct result[4]. In most cases, the algorithm terminates much sooner (i.e., requires much fewer queries) than the deterministic algorithm - only in the worst-case scenario will it be reduced to the deterministic algorithm itself.

Then, in §5, we address the rank-estimation problem for highly ranked tuples. Our key idea here is to first identify all queries which might reveal tuples with higher rank than the input tuple. Then, we select a small subset of these queries in a random yet judicious manner, and issue them to form a COUNT estimation for all higher-ranked tuples - without actually crawling the tuples.

Finally, in §6, we combine the techniques proposed in §4 and §5 to form our final rank estimation algorithm RANK-EST, which can efficiently yet accurately estimate ranks for both highly and lowly ranked tuples. Also in this section, we explain why we aim for rank "estimation" instead of precise "computation" in the paper by proving *hardness results*. The results show that it is not only impossible to precisely compute the rank of a given tuple in an efficient manner, even approximating the rank within a (small) fixed ratio mandates an extremely high query cost in the worst-case scenario.

4. CHALLENGE 1: RANK COMPARISON

In this section, we address the first challenge - rank comparison for proprietary ranking functions. The key task here is to find a sequence of "bridge" tuples connecting the two inputs through direct domination relationships. We start by describing RANK-COMPARE, a deterministic yet inefficient algorithm to solve the problem, identify the fundamental problem underlying its excessive query cost, and address it with LV-RANK-COMPARE, our randomized solution to the problem.

4.1 RANK-COMPARE

Description: The deterministic algorithm starts by testing if t and t' are directly comparable with each other. If not, it finds all tuples which directly dominate t, denoted by $\mathcal{D}(t)$, by issuing all 2^m queries (as mentioned in §2.3) which match t. Then, for each tuple t_i in $\mathcal{D}(t)$, RANK-COMPARE tests if t' directly dominates t_i - if so, then a bridge $t' \succ t_i \succ t$ is found. It does the same for t' - i.e., find $\mathcal{D}(t')$ and test if t dominates any tuple within[5]. If no bridge is found, RANK-COMPARE identifies all tuples directly dominating (at least) one tuple in $\mathcal{D}(t)$ and $\mathcal{D}(t')$, respectively, and uses it to attempt building a two-hop bridge, and (if failed) repeats this process until finding a sequence of bridge-tuples $t_{\mathrm{b}1}, \ldots, t_{\mathrm{b}h}$, such that either $t \succ t_{\mathrm{b}1} \succ \cdots \succ t_{\mathrm{b}h} \succ t'$ or $t' \succ t_{\mathrm{b}1} \succ \cdots \succ t_{\mathrm{b}h} \succ t$. If no such a sequence can be found, then the top-k interface does not reveal enough information for comparing the ranks of t and t'.

In essence, this iterative process is a classic breadth-first-search-based graph reachability algorithm, if we consider all tuples in the hidden database as vertices and the direct domination relationships as edges. Figure 2 demonstrates such a correspondence and an example of building a bridge from t_4 to t_8 in the running example.

Pitfalls of RANK-COMPARE: An obvious problem of RANK-COMPARE is its query cost: in order to find a bridge connecting t with t', the iterative process of RANK-COMPARE repeatedly computes $\mathcal{D}(\cdot)$ which requires numerous queries when m, the number of attributes, is large. The key reason underlying this problem is the way RANK-COMPARE attempts to build the bridge. A large number of queries are wasted testing tuples that are very *unlikely* to be on the bridge, as we shall show in the following example.

Consider the rank comparison between t_8 and t_4 in the running example, which is also illustrated in Figure 2. A key observation

[3]Note that if the input tuple happens to be last-ranked, then this method is reduced to crawling the entire database.

[4]as long as such a result is revealed by the top-k interface - which, according to Theorem 6.1, is highly likely.

[5]We did not pursue the other direction of finding the set of tuples which *are dominated by* t as it may require crawling the entire database - e.g., note that the tuples returned by SELECT * FROM D directly dominate all other tuples.

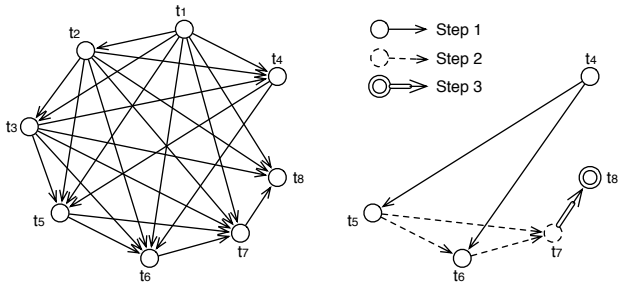

Figure 2: Iteratively build a bridge from t_4 to t_8

here is that $\mathcal{D}(t_8)$ includes not only the tuple that will eventually serve on the bridge (i.e., t_7), but a large number of other tuples (i.e., t_1, t_2, t_3). These tuples tend to be highly ranked - indeed, note that tuples returned by SELECT * FROM D directly dominate any other tuple in the database. As a result, they are highly unlikely to appear on the bridge, especially when t and t' have close yet low ranks, because any tuple which resides on a bridge between t and t' must be ranked between them. Unfortunately, RANK-COMPARE still wastes queries testing these tuples - wasting queries building a bridge-to-nowhere that has surpassed the rank of t'.

In the next subsection, we shall introduce our idea to significantly speed up the bridge-construction process by finding tuples that are most likely to serve on the bridge. Nonetheless, it is important to note that the uncertainty of bridge-construction, i.e., the lack of knowledge on which tuple to select next for building the bridge, is an inherent obstacle for any deterministic rank comparison algorithm. Indeed, we shall prove in §6.3 a negative results which shows that no deterministic algorithm can achieve a realistic worst-case query cost for rank comparison - a motivation for our proposal of a randomized algorithm.

4.2 LV-RANK-COMPARE

In this subsection, we start by describing the overarching scheme of LV-RANK-COMPARE, a *rank-based random walk with restart* on the domination-relationship graph (depicted in Figure 2). Then, we discuss two key design issues for random walk, the selection of the next step and the decision of when to restart the random walk, before presenting the LV-RANK-COMPARE algorithm.

Random Walk For Bridge Construction: To compare the ranks of t and t', we perform the random walk both ways - i.e., we simultaneously start two random walks from t and t', respectively, until one walk reaches the other tuple. Without loss of generality, we consider the walk from t to t'. At each step, we choose a tuple from $\mathcal{D}(t)$, the set of tuples directly dominating t, by issuing a query matching t and finding a higher-ranked tuple from its answer[6]. We repeat this step to form a random walk $t \prec t_{\mathrm{b}1} \prec \cdots \prec t_{\mathrm{b}h}$. For each new tuple $t_{\mathrm{b}i}$ encounter in the walk, we issue one extra query (according to the method in §2.3) to determine if $t' \succ t_{\mathrm{b}i}$ - which indicates the successful construction of a bridge. Otherwise, we either continue or restart the random walk.

There are two key design issues one must address : (1) how to choose the next stop of random walk, $t_{\mathrm{b}i}$, from $\mathcal{D}(t)$, and (2) whether to restart or continue a random walk if a bridge is not (yet) found. We address the two issues respectively as follows.

Selection of $t_{\mathrm{b}i}$: For the first problem, we argue that an ideal selection of $t_{\mathrm{b}i}$ should satisfy the following two conditions:

[6]We shall discuss next how to choose which query to issue and which tuple to select.

α. The rank of $t_{\mathrm{b}i}$ should be as close to $t_{\mathrm{b}i-1}$ as possible, so as to ensure that the bridge does not overpass t'.

β. $t_{\mathrm{b}i}$ should share as many common attribute values with t' as possible, so as to increase the possibility for $t_{\mathrm{b}i}$ to be directly comparable with t'.

One might find Condition α counter-intuitive - rather than trying to make as much "progress" (i.e., rank-increase) as possible towards t', we are seemingly making the "progress" as little as possible. To understand the rationale behind Condition α, we make an important observation on what "progress" really means in the bridge-building process: Note that our objective is to find $t_{\mathrm{b}i}$ which is directly dominated by t'. A key observation here is that, as long as $t_{\mathrm{b}i}$ does not "overshoot" t' (i.e., $t_{\mathrm{b}i}$ has a lower rank), whether $t_{\mathrm{b}i}$ is directly dominated by t' has nothing to do with the rank of $t_{\mathrm{b}i}$, but (instead) is only determined by two factors: (1) the common attribute values shared between $t_{\mathrm{b}i}$ and t', and (2) given the most-specific query matching both $t_{\mathrm{b}i}$ and t', whether t' has a sufficiently high rank to be returned by the query. Since the rank of t' is solely input-dependent, Conditions α and β capture the two objectives under our control: Condition α aims to ensure that $t_{\mathrm{b}i}$ does not overshoot t', while Condition β aims to maximize the probability for $t_{\mathrm{b}i}$ and t' to be directly comparable.

In order to efficiently select $t_{\mathrm{b}i}$ according to the above two conditions, we start by finding queries matching t which are most likely to return tuples with close ranks to t. Specifically, we start with q : SELECT * FROM D, choose predicates matching t (i.e., $A_1 = t[A_1], \ldots, A_m = t[A_m]$) uniformly at random and add one at a time to q until reaching a query q' which returns t. For example, consider the rank comparison between t_4 and t_7 in the running example when $k = 3$. Suppose that, for the random walk from t_7, we happen to choose predicates $A_4 = 0$ and $A_1 = 1$ in order. We will stop at query q': SELECT * FROM D WHERE $A_4 = 0$ AND $A_1 = 1$, because t_7 is not returned by SELECT * FROM D or SELECT * FROM D WHERE $A_4 = 0$.

Then, from the tuples returned by q' which are ranked higher than t, we choose the one which has the most common attribute values with t', and use it as the next stop in the random walk. Note that a special case arises when t is the highest-ranked tuple returned by q'. In this case, we use the parent of q' (by removing the last-added predicate) to find the next stop.

Restart of Random Walks: We now consider the case where the random walk reaches a node $t_{\mathrm{b}i}$ that is not directly dominated by t'. An obvious condition for restarting the random walk is when $t_{\mathrm{b}i}$ indeed directly dominates t' - a clear evidence of overshooting the target. However, if we only restart the random walk in this case, it is still possible for a long (yet wasteful) random walk to continue despite of reaching tuples that far outrank yet are not directly comparable with t'. To address this problem, we make two important observations from real-world experiments: (1) a correct bridge very rarely involves more than a few (e.g., 2) tuples, and (2) there are a large number of such "short bridges". Thus, a more effective approach is to proactively restart a random walk (and hopefully hit one of the many other short bridges) instead of continuing on to a long path that is likely to have already outranked t'.

Based on these two observations, we introduce our strategy of proactively restarting a random walk. A key idea here is that, instead of directly limiting the length of a random walk, we instead place a upper bound c_{N} on the number of *new tuples* involved in a random walk (i.e., which were never included in previous random walks). This ensures the correct discovery of a bridge in the worst-case scenario where even the shortest bridge has a long length. As we shall show in §7, we found through real-world experiments that

$c_N = 3$ is often the optimal setting for hidden databases in practice.

4.3 Algorithm RANK-EST-S

Algorithms 1 and 2 depict the pseudocode of Algorithm LV-RANK-COMPARE and its usage in Algorithm RANK-EST-S, our sampling-based rank estimation algorithm, respectively. Note that in LV-RANK-COMPARE, we simultaneously build bridges from t and t', respectively, in order to enable rank comparison no matter which tuple is ranked higher.

Before concluding the section, we would like to note how LV-RANK-COMPARE compares against RANK-COMPARE. Note that, with the breadth-first scheme, RANK-COMPARE always identifies the *shortest* bridge from t to t'. Such an exhaustive search is unnecessary when *any* valid bridge would suffice. The randomized algorithm LV-RANK-COMPARE instead aims to quickly identify the most likely path from t to t' which can serve as a bridge, thereby significantly reducing the query cost.

Algorithm 1 LV-RANK-COMPARE

Input : t and t', the tuples to be compared
loop
  Set $t_{b0} = t$, $t_{c0} = t'$, $l = 1$
  repeat
    Choose t_{bl} randomly from $\mathcal{D}(t_{bl-1})$
    Choose t_{cl} randomly from $\mathcal{D}(t_{cl-1})$
    return t **if** $t \succ t_{cl}$, or t' **if** $t' \succ t_{bl}$
  until bridges $t_{b0} \dots t_{bl}$ and $t_{c0} \dots t_{cl}$ has c unseen tuples
end loop

Algorithm 2 RANK-EST-S

Input : t , q_F
Take a random sample S from $Sel(q_F)$
n = Estimate of size of D from S
c = number of tuples in S that outrank t
return $\frac{c}{|S|} * n$

5. CHALLENGE 2: HANDLING HIGHLY RANKED TUPLES

We now consider how to enable rank discovery for highly ranked tuples (when the ranking function is either proprietary or observable). We start by describing a deterministic solution RANK-COMPUTE, before introducing a randomized algorithm RANK-EST-H with a significantly lower query cost. As we show in §7, RANK-EST-H is the algorithm of choice for as much as top 25% of the database (after which RANK-EST-S becomes competitive).

5.1 RANK-COMPUTE

Before introducing RANK-COMPUTE, we would like to first note that, while the concept of direct domination relationship was introduced in §2 for proprietary ranking functions, it readily applies to observable ones as well. Even though direct domination relationships (or chains of them) are no longer needed for comparing observable ranks, the concept is still important for understanding how to retrieve tuples that outrank the input, as we show below.

To count the rank of t, we have to consider two types of tuples: those in $\mathcal{D}(t)$ - i.e., directly dominating t - and those that outrank t but are not directly comparable with t, which we denote to as $\mathcal{D}_{\text{i}}(t)$. RANK-COMPUTE crawls all tuples in $\mathcal{D}(t)$ and $\mathcal{D}_{\text{i}}(t)$ in an iterative fashion - i.e., it starts by computing $\mathcal{D}(t)$, followed by computing $\mathcal{D}(\cdot)$ for each tuple in $\mathcal{D}(t)$, and does so iteratively until no new tuple is found. One can see that the key challenge here is to efficiently compute $\mathcal{D}(\cdot)$ - which we address next.

Computing $\mathcal{D}(t)$: As mentioned in §2.3, a baseline method for computing $\mathcal{D}(t)$ is to issue all $\binom{m}{0}+\dots+\binom{m}{m} = 2^m$ queries which match t. To reduce the large query cost of 2^m, we consider a query-issuing strategy described as follows: First, we organize these 2^m queries into a lattice structure - an example of which (when $t = t_7$ in the running example) is shown in Figure 3. One can see that the root of the lattice is `SELECT * FROM D`, while the bottom is the fully specified, m-predicate, query. Each node on Level-i represents a query with i conjunctive predicates (all matching t).

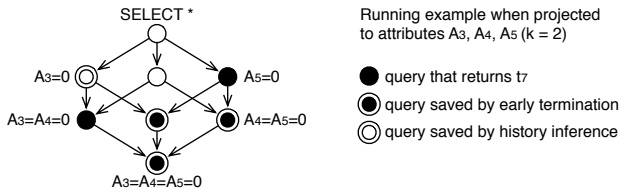

Figure 3: Illustration of Lattice and Two Ideas

Instead of issuing all 2^m queries in the lattice, as in the baseline, we reduce the query cost according to two key ideas. One is *early termination* - i.e., if a query q in the lattice returns t, then we do not need to issue any (lower-level) successors of q because any tuple returned by these queries which ranks higher than t must also be returned by q. In the running example depicted in Figure 3, no descendants of $A_5 = 0$ needs to be issued according to early termination, because $A_5 = 0$ already returns t_7.

The next cost-saving strategy is *history inference*. Note that this includes not only the simple leverage of historic queries - i.e., do not issue a query if it has been issued before - but also the non-trivial inference of a query answer from a collection of historic queries. Specifically, consider the case where, before issuing a query q, we have already obtained answers to all predecessors of q in the lattice (i.e., queries on the path between the root and q) and, among the returned tuples, at least k of them match q. In this case, we do not need to issue q because it is impossible for it to return any tuple we have not yet seen. For example, in Figure 3 we do not need to issue $A_3 = 0$ due to history inference, because the root `SELECT * FROM D` returns t_1 and t_2 which have $A_3 = 0$.

Algorithm 3 RANK-COMPUTE

Input: t; **Output:** $r(t)$
$H(t)$ = { $\mathcal{D}(t)$ } (set of tuples ranked higher than t)
repeat $H(t) = H(t) \cup \mathcal{D}(t')\ \forall t' \in H(t)$
until no new tuples added to $H(t)$
return $|H(t)| + 1$

Leveraging both ideas, RANK-COMPUTE collects $\mathcal{D}(t)$ by performing a breadth-first search (BFS) of the lattice. In the design of lattice-BFS, we skip any query that can be inferred from history, and invoke early termination if reaching a node that returns t.

Theoretical Analysis: Algorithm 3 depicts the pseudocode of RANK-COMPUTE. The following theorem provides an upper bound on the query cost of RANK-COMPUTE.

THEOREM 5.1. *The worst-case query cost for RANK-COMPUTE to compute $r(t)$, i.e., the rank of a tuple t, is $r(t) \cdot \binom{m}{\lfloor m/2 \rfloor}$.*

We do not include the proof of this theorem due to space limitations. Note that, for computing $\mathcal{D}(t)$, RANK-COMPUTE requires

Algorithm 4 RANK-EST-H

1: **Input :** t , q_F
2: Randomly drill down on query lattice augmented with q_F for t until some query node N_q returns t
3: $\mathbb{D}(t)$ = set of tuples returned by ancestors(N_q) and dominate t
4: **return** $\sum_{t_i \in \mathbb{D}(t)} \frac{1}{p(t_i)}$

at most $\binom{m}{m/2} = O(2^{\sqrt{m} \cdot \log m/2})$ queries - a significant reduction from the 2^m queries required by the baseline method. For example, when $m = 12$, the query-cost is reduced from 4,096 to 924.

5.2 RANK-EST-H: A Randomized Algorithm

Even though RANK-COMPUTE requires much fewer queries than the baseline method, it can still generate excessive query cost when either m or $r(t)$ is large. For example, the upper bound derived in Theorem 5.1 exceeds 25,200 queries for a 100-th ranked tuple in a 10-attribute database. This can be attributed to : (1) the iterative process for computing $\mathcal{D}_{\mathrm{i}}(t)$ (tuples that outrank but are not directly comparable with t), as many tuples are repeatedly retrieved in this iterative process, and (2) the actual crawl of all tuples in $\mathcal{D}(t)$ and $\mathcal{D}_{\mathrm{i}}(t)$, when only COUNT is required by rank estimation. We address these two problems respectively as follows.

For the computation of $\mathcal{D}_{\mathrm{i}}(t)$, a key observation here is that the higher ranked t is, the smaller $|\mathcal{D}_{\mathrm{i}}(t)|$ is likely to be. To understand why, recall from §2 that in order for two tuples t and t' to be *not* directly comparable with each other, the most concrete query q which matches both tuples must not return either of them. One can see that clearly, the higher ranked t and t' are, the less likely it is for q to return neither of them. Thus, $|\mathcal{D}_{\mathrm{i}}(t)|$ is likely to be small for a highly ranked input tuple t. This can be observed from the running example - for the highest ranked 6 tuples $t_1, \ldots, t_6$, $\mathcal{D}_{\mathrm{i}}(\cdot)$ are all zero. On the other hand, the 8-th ranked tuple t_8 has 3 tuples (i.e., t_4, t_5, t_6) in $\mathcal{D}_{\mathrm{i}}(t_8)$.

This key observation leads us to a simple yet (somewhat) crude way of estimating $r(t)$: crawl the lattice structure to retrieve all tuples in $\mathcal{D}(t)$, and then use $|\mathcal{D}(t)|$ as an estimation for $r(t)$. Nonetheless, the second problem (i.e., crawling $\mathcal{D}(t)$ when only COUNT is required) remains. To address this problem, our main idea is to enable an efficient estimation of $|\mathcal{D}(t)|$ by performing a *random drill-down* from the top of the lattice as depicted in Figure 4 - i.e., starting from the root, we choose a branch uniformly at random, and then repeat this process to "drill down" deeper into the lattice in order to sample each tuple dominating t with a positive probability. Figure 4 depicts examples of drill downs for the lattice of t_5 in the running example when projected to attributes A_1, A_2, A_3.

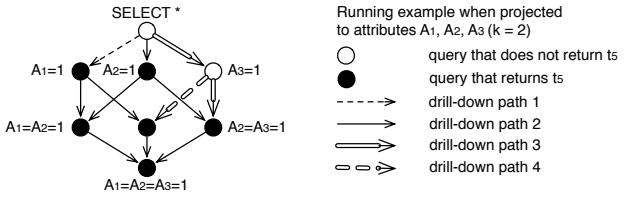

Figure 4: Examples of Drill Downs

Let $t_1, \ldots, t_w$ be the tuples retrieved during this drill down which directly dominate t. One can see that, by applying the Horvitz-Thompson estimator [12], an unbiased estimate for $|\mathcal{D}(t)|$ is $\sum_{i=1}^{w} 1/p(t_i)$, where $p(t_i)$ is the probability for t_i to be picked up by such a random drill-down process, because its expected value

$$E\left[\sum_{i=1}^{w} \frac{1}{p(t_i)}\right] = \sum_{t' \in \mathcal{D}(t)} \left(p(t_i) \cdot \frac{1}{p(t_i)}\right) = |\mathcal{D}(t)|. \quad (2)$$

Unfortunately, computing $p(t_i)$ proves to be challenging because a tuple directly dominating t may be returned by multiple drill-down paths. For example, in Figure 4, t_4 in the running example may be returned by drill down paths 2, 3 and 4. In addition, note that different drill-down paths are taken with different probability - e.g., while path 2 in Figure 4 is taken with probability 1/3, path 3 is take with only 1/6 probability. As such, one may not be able to precisely compute $p(t_i)$ without issuing additional queries after the drill-down process.

To address this challenge, we consider the following heuristics: if, during the current drill down, tuple t_i is first returned at Level-h of the lattice[7], then we assume that all other nodes at Level-h which match t_i also return t_i - and no node above Level-h returns it. With this heuristics, we now compute an estimation of $p(t_i)$ as follows. Note that the probability for the random drill-down process to reach an h-th level node is $1/\binom{m}{h}$. Thus, for a tuple t_i which shares the same value as t on c attributes (note that $c \geq h$ due to the lattice definition), we have $p(t_i) \approx \binom{c}{h} / \binom{m}{h}$.

For example, in Figure 4, when we retrieve t_4 at the Level-1 node $A_2 = 1$, the estimate[8] of $p(t_4) \approx \binom{2}{1}/\binom{3}{1} = 2/3$. One can see that we can now estimate $|\mathcal{D}(t)|$ accordingly. For example, if we happen to take drill down path 2 in Figure 4, our estimation for $\mathcal{D}(t_5)$ is $\mathcal{D}(t) \approx 2/1 + 1/(2/3) = 3.5$, leading to a rank estimation of 4.5 for $r(t_5)$. Note that we may repeat the random drill down for multiple times (and take the average of estimations) to improve the estimation accuracy for $|\mathcal{D}(t)|$.

6. ALGORITHM RANK-EST

In this section, we start by describing our final RANK-EST algorithm. Then, we tackle two theoretical issues, (1) the possible discrepancy between real and revealed ranks, and (2) the hardness of exact rank computation, respectively.

6.1 Description of RANK-EST

Interleaving: From the previous discussions, one can see that the RANK-EST-S and RANK-EST-H have complementary behavior - i.e., RANK-EST-S works poorly for highly ranked tuples, which RANK-EST-H specifically address. We now consider the integration of these two algorithms to produce RANK-EST which work universally for all tuples in the database. Our main idea is to *interleave* the two algorithms. In particular, we first take a pilot sample of the hidden database and use RANK-EST-S to produce a (roughly) estimated rank. If the confidence interval of the estimation falls below a threshold[9], then t likely has a high rank - thus we switch to RANK-EST-H. Otherwise, we continue with the sampling process in RANK-EST-S to reduce the estimation error.

Extension to other q_{F}: So far, we focused on the case where the user-specified filtering query q_{F} in Definition 2 is `SELECT * FROM D`. We now consider the extension to other q_{F} for RANK-EST-S and RANK-EST-H, respectively. Note that, for RANK-EST-S, there is indeed no revision required for handling other q_{F}, as

[7] Let the root level be Level 0.

[8] where $c = 2$, $m = 3$, and $h = 1$. Note that $m = 3$ because the lattice represents a projection to A_1, A_2, A_3. $c = 2$ because t_4 and the input t_5 have two attributes in common among A_1, A_2, A_3.

[9] e.g., for a 95% confidence interval $r(t) \in [u, v]$, if $v < \ell$ where ℓ is the pre-determined threshold

long as the sampling algorithm we call as a subroutine only generates samples that satisfy q_{F}. Then, by calling on existing aggregate estimation algorithms (e.g., [8]) to estimate the COUNT of tuples satisfying q_{F}, we can readily generate the rank of a tuple within q_{F}.

For RANK-EST-H, a simple revision is required - in Algorithm 4, when computing $\mathbb{D}(t)$, i.e., the set of tuples returned during drill-down which dominate t, we only include those tuples in $\mathbb{D}(t)$ that satisfy q_{F}. With the revision, the estimation we generate is for the number of tuples in $\mathcal{D}(t)$ matching q_{F}. Note that, if q_{F} happens to be a conjunctive query that can be specified through the top-k interface, then it is possible to further improve the efficiency of RANK-EST-H by *appending* the predicates in q_{F} to every query in the lattice (on which we perform the random drill-downs, as shown in Figure 4). This way, all queries issued by RANK-EST-H are focused on tuples matching q_{F}, leading to a reduced query cost. Algorithm 5 depicts the interleaved RANK-EST generic to q_{F}.

Algorithm 5 RANK-EST

1: **Input :** t , q_F **Output :** $\tilde{r}(t)$
2: Fetch pilot sample S from $Sel(q_F)$
3: $\tilde{r_c}(t)$ = Approximate rank estimated by RANK-EST-S(t, q_F)
4: **if** confidence interval of $\tilde{r_c}(t) < threshold$ **then** Estimate $\tilde{r}(t)$ through RANK-EST-H(t, q_F)
5: **else** Continue estimation of $\tilde{r}(t)$ through RANK-EST-S(t, q_F)

6.2 Closeness of Real and Revealed Ranks

To understand why a tuple's true rank might differ from what is revealed through the top-k interface, consider a database of two Boolean attributes and three tuples: $t_1 : \langle 0, 1\rangle$, $t_2 : \langle 0, 0\rangle$, and $t_3 : \langle 1, 1\rangle$, with t_1 and t_3 having the highest and lowest rank, respectively, according to a hidden ranking function. One can see that, if the database has a top-1 interface, then it is impossible to determine which of t_2 and t_3 ranks higher, because the only query that matches both tuples, i.e., `SELECT * FROM D`, returns neither. This example illustrates that, while the true rank of t_3 should be 2 (because two tuples rank higher than it), the best estimation one can make from the top-k interface is 1 - leading to a significant difference between the two values.

Fortunately, we found through theoretical analysis and experimental studies that such a difference is usually extremely small to non-existent in real-world hidden databases, mainly because of two reasons. First, real-world databases often feature a much larger k (than 1), revealing a lot more information about the rank comparison between different tuples. Second, there are often many more attributes, making it unlikely for two highly ranked tuples to be incomparable with each other. The following theorem uses a special case to illustrate the extremely small value of the difference. We shall further evaluate such a difference value with real-world datasets in the experiments section.

THEOREM 6.1. *Consider a database with m attributes, each of which is generated i.i.d. with uniform distribution over a domain size of c. For a tuple t with real rank r and top-k-interface-revealed-rank r', there is $\Pr\{|r - r'| > \epsilon\} < p \cdot (2p)^{\epsilon-1}$, where p is upper-bounded by: (note that $\text{erf}(\cdot)$ is the error function)*

$$\sum_{i=0}^{m}\left[\binom{m}{i}\cdot\frac{(c-1)^{m-i}}{c^m}\cdot\left(1-\sum_{j=0}^{k-1}\binom{r}{j}\cdot\frac{(c^i-1)^{r-j}}{c^{i\cdot r}}\right)\right]$$
$$\leq\sum_{i=0}^{m}\left[\frac{\binom{m}{i}\cdot(c-1)^{m-i}}{2c^m}\cdot\left(1-\text{erf}\left(\frac{(k-1)\cdot c^{i/2}-r}{c^i\sqrt{2r(c^i-1)}}\right)\right)\right]$$

We do not include the proof here due to space limitations. One can see from the theorem that, the larger k is or the smaller m and r are, the smaller the probability for $|r - r'| > \epsilon$ will be - irrelevant of how many tuples the database contains. Specifically, for a 15-attribute database with $k = 100$, the largest (i.e., worst-case) probability for any tuple (with arbitrary r) to have a relative rank difference of 2% (i.e., $|r-r'| > 0.02\cdot r$) is lower than 0.00058 - indicating that the top-k interface likely reveals a very accurate rank for each tuple in the database.

6.3 Hardness Results

We now consider the hardness of rank comparison and computation for a hidden web database. Recall the query lattice defined in §4. When there is a large number of attributes in the database, the query cost of rank comparison/computation can be very high if it requires the enumeration of queries at the middle level (i.e., $\lfloor m/2\rfloor$-th level) of the lattice - because this middle level contains the most queries (i.e., $\binom{m}{m/2}$). To understand how rank comparison/computation may require issuing such middle-level queries, consider an example where one needs to compute the rank of a given tuple t which has such a low rank that it is not returned by any query above the $\lfloor m/2\rfloor$-th level in the lattice. Note that, unfortunately, this scenario can happen even when the database is very small - with as few as $m/2 + 1$ tuples, as we shall show in the proof. In this case, to determine if there is a tuple t' which (1) shares the values of $A_1, \ldots, A_{m/2}$ with t and (2) directly dominates (or is directly dominated by) t, one has no choice but to issue `SELECT * FROM D WHERE` $A_1 = t[A_1]$ `AND` $\cdots$ `AND` $A_{m/2} = t[A_{m/2}]$ because:

- any query with fewer predicates would not help to determine if $t \succ t'$, because t is not returned by such a query
- any query with more predicates would not help because it does not match t'.

Intuitively, one can see that such a scenario may force the issuing of all queries in the middle level of the lattice.

THEOREM 6.2. **(Hardness of Rank Comparison)** *Given two comparable tuples t and t', no algorithm can guarantee correct comparison with $o(m^{\sqrt{m}/2})$ queries.*

We do not include proof here due to space limitations. One can see that the theorem precludes the existence of efficient deterministic algorithms for rank comparison when there is a large number of attributes (e.g., when $m^{\sqrt{m}/2} \gg n$). The following corollaries further eliminate the possibility of having a worst-case-efficient algorithm for computing the exact rank of a given tuple, or even approximating the rank with a deterministic error bound, when there is a large number of attributes in the database and/or the attribute domain sizes are unbounded. However we show in §7 that our randomized algorithms LV-RANK-COMPARE and RANK-EST usually requires far fewer queries for real-world datasets.

COROLLARY 6.2.1. **(Hardness of Obtaining the Exact Rank)** *For a given tuple t, no algorithm can guarantee the computation of the rank of t with $o(m^{\sqrt{m}/2})$ queries. If the domain sizes of attributes are sufficiently large, no algorithm can guarantee rank computation with $o(m^{\sqrt{m}/2} + n^n)$ queries.*

COROLLARY 6.2.2. **(Hardness of Rank Approximation)** *For a given tuple t, no algorithm can generate a value v such that the rank of t is guaranteed to be in $[v, v \cdot r]$ with $o(m^{\sqrt{m}/2} - \sqrt{m} \cdot r)$ queries. If the domain sizes of attributes are sufficiently large, no algorithm can do so with $O(m^{\sqrt{m}/2} + n^n - \sqrt{m} \cdot r)$ queries.*

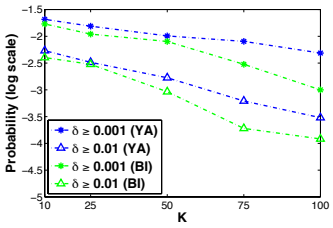

Figure 5: Rank Discrepancy

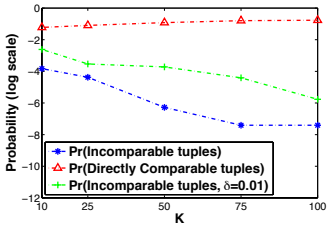

Figure 6: Comparability of Tuples

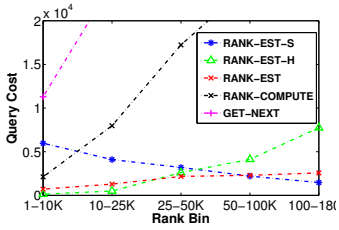

Figure 7: Varying Input Rank (RE)

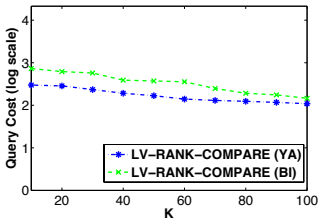

Figure 8: Varying k (RC)

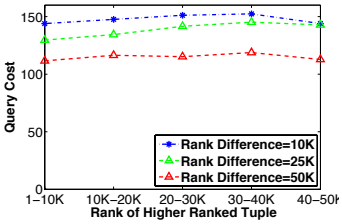

Figure 9: Varying Input Rank (RC)

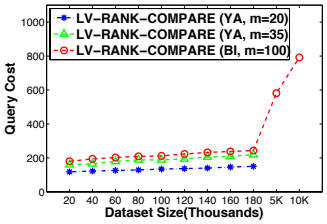

Figure 10: Varying n (RC)

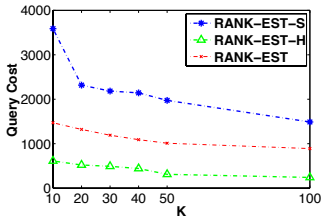

Figure 11: Varying k (RE)

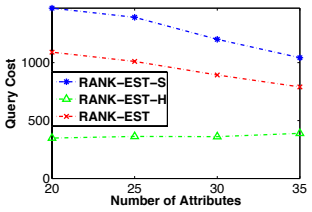

Figure 12: Varying m (RE)

7. EXPERIMENTAL RESULTS

7.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a quad-core 2 GHz AMD Phenom machine with 8 GB of RAM. The algorithms were implemented in Python.

Offline Datasets: Our primary dataset consists of data crawled from the Yahoo! Autos (YA)[10], a real-world hidden database. 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, etc whose domain size range from 5 to 16. We also tested our algorithms over a synthetic boolean dataset (BI) of 10 million tuples and 100 attributes for scalability purposes. The tuples are generated as i.i.d. data with each attribute having probability of $p = 0.5$ to be 1. Since neither dataset comes with a ranking function, we produce a synthetic static ranking for each dataset by first generating a random permutation of all tuples in a dataset, and then use each tuple's position in the permuted list as its rank. Default value for k is 100. Our charts primarily report the results over for Yahoo! Autos (unless otherwise specified).

Online Real-World Experiment: We also tested our algorithms online via Amazon.com's Product Advertising API[11] that reveals a top-100 interface where the items are ordered by a proprietary ranking function based on sales rank. The actual rank is not revealed by default. To uncover the ground truth, we found that the individual item description provided by Amazon.com reveals the sales rank of certain items[12]. As such, we chose all testing tuples from those that have its real rank disclosed in the description. Specifically, we focused on Amazon's DVD and book items, and constructed search queries using 15 categorical attributes such as Actor, Artist, etc. Amazon.com has a limit of 2,000 queries per IP address per hour.

Algorithms: We tested 4 algorithms for rank estimation considered in the paper: RANK-COMPUTE, RANK-EST-S, RANK-EST-H and RANK-EST. Since, RANK-EST-S uses algorithm LV-RANK-COMPARE for rank comparison, we also perform a rigorous set of experiments to evaluate it. For RANK-EST-S and RANK-EST, we used the existing HIDDEN-DB-SAMPLER [7] for the sampling primitive. We also tested as baseline the direct usage of an existing algorithm GETNEXT [19] for rank computation. Since GETNEXT is only capable of obtaining the $(h + 1)$-th ranked tuple based on the top-h tuples, to use it for rank computation we ran the algorithm repeatedly until reaching the input tuple.

Performance Measures: For all algorithms, we measure efficiency through query cost. In addition, we measure the accuracy of rank estimation through the *relative error* measure defined in §2.4.

7.2 Experimental Results

In this subsection, we first empirically evaluate the discrepancy between real and revealed ranks. Then we describe the results of our offline experiments over Yahoo! Auto dataset and online experiments at Amazon.com for the rank estimation problem.

Empirical Evaluation of Rank Discrepancy and Tuple Comparability : Recall that our aim is to estimate the revealed rank of a given tuple. We observed that problem of discrepancy between real and revealed is exceedingly unlikely in practice. For each tuple in both datasets, we computed its revealed rank using RANK-COMPUTE algorithm and compared the relative error between the real and revealed ranks. Figure 5 shows the fraction of tuples that had a rank discrepancy of 0.1% and 1% (or above) for different values of k. For $k = 100$ (a fairly common value), less than 400 tuples out of 200,000 tuples had a rank discrepancy of 1%. This justifies our problem definition in terms of revealed rank and that rank discrepancy is not a big issue in real-world databases.

Figure 6 shows the feasibility of rank comparison problem over Yahoo! Autos dataset. The results for Bool-IID dataset was similar. Note that the probability of finding a pair of tuples that are not comparable is exceedingly low ($\approx 10^{-7}$ for $k = 100$). Further, the fraction of tuple pairs that are directly comparable varies between 10-20% and increases with larger values of k. It shows that while almost all pair of tuples are comparable only a fraction of them are directly necessitating practical algorithms. Finally, we identify the fraction of pairs of tuples that have a relative rank difference of 1% (e.g. when tuples ranked 990th to 1010th were compared with the 1000th-ranked tuple). Intuitively, these are the hardest pairs to test as with larger rank difference, the chance of identifying a bridge dramatically increases. We observed that even for this restrictive scenario, only a minuscule fraction of tuples remain incomparable.

[10] http://auto.yahoo.com

[11] https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html

[12] Many others, e.g., item No. B009B0JR2C, do not reveal the rank at all.

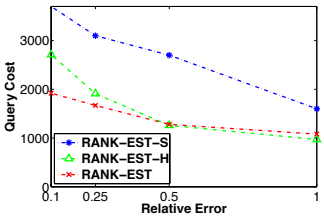

Figure 13: Tradeoff (RE)

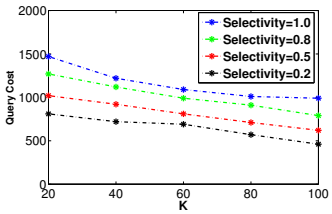

Figure 14: Varying Selectivity of q_F

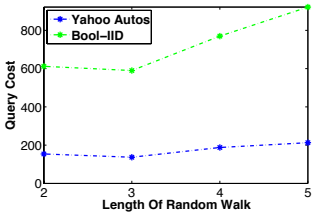

Figure 15: Varying c_N (RC)

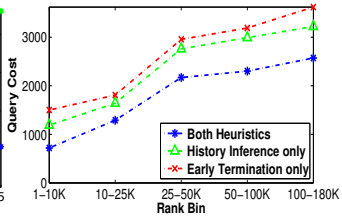

Figure 16: Impact of Heuristics on Query Cost (RE)

Comparison of Rank Estimation Algorithms: We start by comparing our algorithms with GETNEXT [19] while varying the rank of the input tuple between 1 and 180,000. Figure 7 depicts the (average) query cost required to achieve a relative error of 10% on rank estimation. The query cost of GETNEXT is only plotted for input rank [1, 10000] as it becomes prohibitive for lower ranked tuples. We also evaluated our deterministic algorithm RANK-COMPUTE. While this algorithm is much more efficient that GETNEXT, its query cost rapidly increases with the tuple's rank and becomes prohibitive. One can also observe from the figure that, as discussed in §5, RANK-EST-S works better for lowly ranked tuples, while RANK-EST-H works better for highly ranked ones. By interleaving the two, RANK-EST works effectively for tuples of all ranks. Given the excessive query cost of RANK-COMPUTE, we only focus on the practical algorithms RANK-EST, RANK-EST-S and RANK-EST-H for comparative analysis.

Rank Comparison: We start by evaluating the efficacy of of our Las Vegas algorithm, LV-RANK-COMPARE. Specifically, we randomly selected two tuples with rank between 42,000 and 44,000, and measured the query cost of comparing them while varying k between 10 and 100. One can see from Figure 8 that, LV-RANK-COMPARE is extremely practical and identifies the correct comparison rapidly. Notice that the query cost decreases rapidly as k increases as more and more tuples becomes directly comparable.

We tested the performance of LV-RANK-COMPARE while varying the ranks of input tuples. Figure 9 depicts the results when the rank difference between the two input tuples varies from 10,000 to 50,000, while the higher-ranked tuple is randomly selected from one of the five rank-buckets ranging from [1, 10000] to [40001, 50000]. One can see from the figure that, consistent with intuition, our algorithm requires fewer queries when the ranks of input tuples are further apart. In addition, the performance of our algorithm is not significantly affected by the absolute rank of the input tuples.

We also tested the scalability of LV-RANK-COMPARE by varying n, the number of tuples, and m, the number of attributes. To do so, we sample tuples and attributes uniformly at random (without replacement) from both datasets. One can see from Figure 10 that our query cost increases slowly with n. Note that the jump at the right side of the figure is because we include at the end of x axis the results for Bool-IID when the dataset contains 5 or 10 million tuples - demonstrating the scalability of LV-RANK-COMPARE. In addition, our query cost actually *decreases* with a large m. The reason for the latter is that, when m is larger, the number of tuples directly comparable with t also increases - leading to a higher probability of bridge construction by LV-RANK-COMPARE.

Rank Estimation: Similar to Figures 8 and 10 for rank comparison, we tested the performance of our rank estimation algorithms against varying k, n and m. All these figures depict the number of queries required for reaching a relative error of 10% for rank estimation. For Figure 12, we randomly chose the input tuples for RANK-EST-H and RANK-EST-S from rank-bucket [10K, 20K] and [50K, 100K], respectively. For RANK-EST, we randomly chose the input from the entire database. From Figures 11 and 12 that, our algorithms require fewer queries when k or m is larger. Figure 13 further depicts the tradeoff between query cost and the relative error of rank estimation. We also tested the impact of selectivity of different queries q_F. We constructed the filtering queries by first randomly deciding the total number of attributes in q_F which (along with their values) are then chosen randomly. We then chose an arbitrary tuple from $Sel(q_F)$ and estimated its rank within it. The results for different queries with varying level of selectivity are provided in Figure 14. As expected, when queries become highly selective, the query cost to estimate its rank drops.

Impact of Heuristics: We now compare the effectiveness of the heuristics used in our algorithms based on their impact over query cost. Note that the accuracy of estimated ranks is not affected by the heuristics being used, because the heuristics are used inside the LV-RANK-COMPARE subroutine which always produces the correct rank comparison. The heuristics that are used inside RANK-EST (early termination and history inference), reduce the query cost by allowing us to compute the output of some queries locally from previously executed query results. Thus, all our heuristics reduce the query cost without affecting the output of our algorithms.

For LV-RANK-COMPARE, we tested our parameter setting for c_N - i.e., the upper bound on the length of a random walk before triggering proactive restart. Figure 15 shows the justification for our heuristics of c_N = 3 for both datasets. We observed similar result in our online experiments also. As discussed in §5, further increasing c_N leads to a higher query cost because, when the random walk "overshoots" the destination, it takes longer to restart before finding one of the many short bridges.

For RANK-EST, we tested how the heuristics - early termination and history inference affect the query cost. We can observe from Figure 16 that both heuristics result in a substantial reduction of query cost. For highly ranked tuples, the impact is more pronounced as RANK-EST-H uses them extensively.

Online Experiments at Amazon.com: Before presenting the results of our online experiments, we would like to note that Amazon's interface provides no efficient way to crawl a large number of lowly ranked tuples without exceeding the query allowance. As such, for the purpose of our experiments, we focused on tuples with rank between 1 and 15,000. Figure 17 shows the results. We first tested our LV-RANK-COMPARE algorithm by randomly selecting two tuples with rank difference varying between 1,000 and 5,000. We can see that our algorithms require fewer than 100 queries for rank comparison. For rank estimation, we can observe that our RANK-EST algorithm requires fewer than 400 queries - far below the hourly limit of 2,000 queries imposed by Amazon - to reach an estimation error of 10%. In addition, the pattern of query-cost change with input rank is consistent with the offline case. Interestingly, RANK-EST-H consistently outperforms RANK-EST-S in

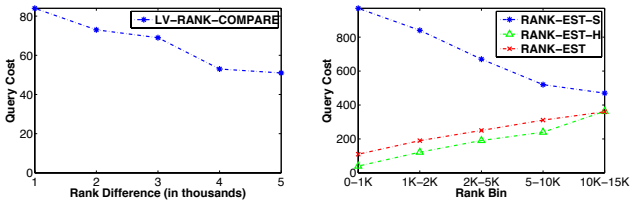

Figure 17: Rank Comparison and Estimation at Amazon.com

our online experiments - indicating that the rank-bucket [1, 15000] we were able to use is still (relatively) highly ranked in the large Amazon database. It is also important to note that our final RANK-EST algorithm only has a slightly higher query cost than RANK-EST-H, indicating our algorithm's ability to quickly switch from RANK-EST-S when the input tuple is highly ranked.

8. RELATED WORK

Data Analytics over Hidden Databases: There has been prior work on crawling, sampling, and aggregate estimation over the hidden web, specifically over text [3, 4] and structured [16] hidden databases and search engines [15, 18, 2]. Specifically, sampling-based methods were used for generating content summaries [6, 14, 11], processing top-k queries [5], etc. Prior work (see [8] and references therein) considered sampling and aggregate estimation over attribute values *explicitly* returned by the web interface of the structured hidden database. Our work, on the other hand, considers rank information of tuples which is *not explicitly* returned, thereby precluding the applicability of prior work. Our paper differs from top-k processing (see [13] for a survey) as our paper aims to discover rank-related information from the top-k answers, instead of studying how th e top-k answers can be retrieved.

Retrieving Rank Information from Hidden Databases: GETNEXT operator [19] allows an ordered crawling of top ranked tuples for any static ranking function. Specifically, given all tuples ranked in top-(h-1) as input, GETNEXT operator retrieves the No. h ranked tuple. While [19] solves the problem of retrieving top-h tuples over a top-k interface (where $h > k$), our paper initiates the first formal study on efficiently estimating the rank of any given tuple in a hidden web database. Admittedly, it is possible to use GETNEXT in a brute-force way to solve the rank computation problem - i.e., by retrieving all tuples ranked higher than the input tuple in an iterative fashion. However, such an approach is prohibitively expensive for all but the highly ranked tuples. Further, it produces an *exact* rank whereas for a number of scenarios as outlined in §1, *approximate* rank with low query cost is preferable. *Suggestion sampling* [3] estimates the frequency of search queries from ranked lists returned by a search engine's prefix-matching auto-complete interface. Unlike [3], we consider the retrieval of rank information from a structured hidden database with a form-like interface.

9. FINAL REMARKS

In this paper, we defined a novel problem of rank discovery from hidden web databases with restrictive top-k search interfaces. We first introduced a taxonomy of ranking functions according to multiple dimensions, discussed the feasibility of rank discovery for each type of ranking function, and described solutions for all the feasible types. We proposed RANK-EST, a randomized algorithm for efficient rank discovery for proprietary and observable ranking functions. We proved hardness results that preclude the existence of efficient deterministic algorithms. We demonstrated the effectiveness of our proposed algorithms using real-world datasets and also through an online experiment conducted over Amazon.com.

10. REFERENCES

[1] F. N. Afrati, P. V. Lekeas, and C. Li. Adaptive-sampling algorithms for answering aggregation queries on web sites. *DKE*, 64(2):462–490, 2008.

[2] Z. Bar-Yossef and M. Gurevich. Efficient search engine measurements. In *WWW*, pages 401–410, 2007.

[3] Z. Bar-Yossef and M. Gurevich. Mining search engine query logs via suggestion sampling. In *VLDB*, pages 54–65, 2008.

[4] K. Bharat and A. Broder. A technique for measuring the relative size and overlap of public web search engines. In *WWW*, pages 379–388, 1998.

[5] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.

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

[7] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *SIGMOD*, pages 629–640, 2007.

[8] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das. Unbiased estimation of size and other aggregates over hidden web databases. In *SIGMOD*, pages 855–866, 2010.

[9] A. Dasgupta, N. Zhang, and G. Das. Leveraging count information in sampling hidden databases. In *ICDE*, pages 329–340, 2009.

[10] A. Dasgupta, N. Zhang, and G. Das. Turbo-charging hidden database samplers with overflowing queries and skew reduction. In *EDBT*, pages 51–62, 2010.

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

[12] D. Horvitz and D. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47:663–685, 1952.

[13] I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40:11:1–11:58, 2008.

[14] P. Ipeirotis and L. Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *VLDB*, pages 394–405, 2002.

[15] K. Liu, C. Yu, and W. Meng. Discovering the representative of a search engine. In *CIKM*, pages 652–654, 2002.

[16] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, pages 129–138, 2001.

[17] C. Sheng, N. Zhang, Y. Tao, and X. Jin. Optimal algorithms for crawling a hidden database in the web. In *VLDB*, pages 1112–1123, 2012.

[18] M. Shokouhi, J. Zobel, F. Scholer, and S. Tahaghoghi. Capturing collection size for distributed non-cooperative retrieval. In *SIGIR*, pages 316–323, 2006.

[19] S. Thirumuruganathan, N. Zhang, and G. Das. Breaking the top-k barrier of hidden web databases. In *ICDE*, pages 1045–1056, 2013.

[20] F. Wang and G. Agrawal. Effective and efficient sampling methods for deep web aggregation queries. In *EDBT*, pages 425–436, 2011.