Mining Search Engine Query Logs via Suggestion Sampling

Ziv Bar-Yossef[*]
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
and
Google Haifa Engineering Center, Israel
zivby@ee.technion.ac.il

Maxim Gurevich[†]
Dept. of Electrical Engineering
Technion, Haifa 32000, Israel
gmax@tx.technion.ac.il

ABSTRACT

Many search engines and other web applications suggest auto-completions as the user types in a query. The suggestions are generated from hidden underlying databases, such as query logs, directories, and lexicons. These databases consist of interesting and useful information, but they are typically not directly accessible.

In this paper we describe two algorithms for sampling suggestions using only the public suggestion interface. One of the algorithms samples suggestions uniformly at random and the other samples suggestions proportionally to their popularity. These algorithms can be used to mine the hidden suggestion databases. Example applications include comparison of popularity of given keywords within a search engine's query log, estimation of the volume of commercially-oriented queries in a query log, and evaluation of the extent to which a search engine exposes its users to negative content.

Our algorithms employ Monte Carlo methods in order to obtain unbiased samples from the suggestion database. Empirical analysis using a publicly available query log demonstrates that our algorithms are efficient and accurate. Results of experiments on two major suggestion services are also provided.

1. INTRODUCTION

Background and motivation. In an attempt to make search experience easier and faster, search engines and other interactive web sites suggest users auto-completions of their queries. Figure 1 shows the suggestions provided by the

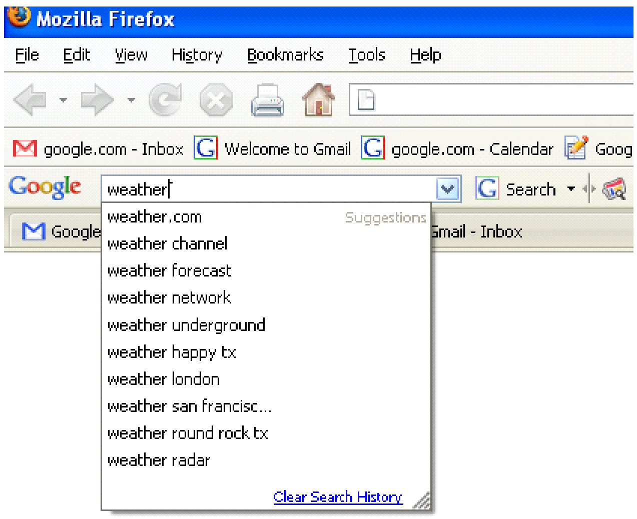


Figure 1: Google Toolbar Suggest.

Google Toolbar[1] when typing the query "weather". Other suggestion services include Yahoo! Search Assist[2], Windows Live Toolbar[3], Ask Suggest[4], and Answers.com[5].

A suggestion service accepts a string currently being typed by the user, consults its internal index for possible completions of this string, and returns the user the top k completions. The internal index is based on some underlying database, such as a log of past user queries, a dictionary of place names, a list of business categories, or a repository of document titles. Suggestions are usually ranked by some query-independent scoring function, such as the popularity of suggestions in the underlying database.

Suggestion databases frequently contain valuable and interesting information, to which the suggestion interface is the only public access point. This is especially true for search query logs that are kept confidential, due to user privacy constraints. Extracting aggregate, non-specific, information from hidden suggestion databases could be very

[*]Supported by the Israel Science Foundation and by the IBM Faculty Award.

[†]Some of this work was done while the author was at the Google Haifa Engineering Center, Israel.

[1]`toolbar.google.com`.
[2]`search.yahoo.com`.
[3]`toolbar.live.com`.
[4]`www.ask.com`.
[5]`www.answers.com`.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

useful for online advertising, for evaluating the quality of search engines, and for user behavior studies. We elaborate below on the two former applications.

Online advertising and keyword popularity estimation. In online advertising, advertisers bid for search keywords that have the highest chances to funnel user traffic to their websites. Finding which keywords carry the most traffic requires access to the hidden search engine's query log. The algorithms we describe in this paper can be used to estimate the popularity of given keywords. Using our techniques, an advertiser who does not have access to the search engine's logs, would be able to compare alternative keywords and choose the ones that carry the most traffic. By applying our techniques continuously, an advertiser can track the popularity of her keywords over time.[6]

Search engine evaluation and ImpressionRank sampling. External and objective evaluation of search engines is important for enabling users to gauge the quality of the service they receive from different search engines. Example metrics of search engine index quality include the index's coverage of the web, the freshness of the index, and the amount of negative content (pornography, virus-contaminated files, spam) included in the index. The limited access to search engines' indices via their public interfaces make the problem of evaluating the quality of these indices very challenging. Previous work [5, 3, 6, 4] has focused on generating random uniform sample pages from the index. The samples have then been used to estimate index quality metrics, like index size and index freshness.

One criticism of this evaluation approach is that it treats all pages in the index equally, regardless of whether they are actually being actively served as search results or not. To overcome this difficulty, we define a new measure of "exposure" of pages in a search engine's index, which we call "ImpressionRank". Every time a user submits a query to the search engine, the top ranking results receive "impressions" (see more details in Section 7 how the impressions are distributed among the top results). The ImpressionRank of a page x is the (normalized) amount of impressions it receives from user queries in a certain time frame. By sampling queries from the log proportionally to their popularity and then sampling results of these queries, we can sample pages from the index proportionally to their ImpressionRank. These samples can then be used to obtain more sensible measurements of index quality metrics (coverage, freshness, amount negative content), in which high-exposure pages are weighted higher than low-exposure ones.

Our techniques can be also used to compare different suggestion services. For example, we can find the size of the suggestion database, the average number of words in suggestions, or the fraction of misspelled suggestions.

Our contributions. We propose algorithms for sampling and mining suggestions using only the public suggestion interface. Specifically, our algorithms can be used either to *sample* suggestions from the suggestion database according to a given *target distribution* (e.g., uniform or popularity-induced) or to compute *integrals* over the suggestions in the database relative to some target measure. Such primitives can then be used to accomplish various mining tasks, like the ones mentioned above.

We present two sampling/mining algorithms: (1) an algorithm that is suitable for uniform target measures (i.e., all suggestions have equal weights); and (2) an algorithm that is suitable for popularity-induced distributions (i.e., each suggestion is weighted proportionally to its popularity). Our algorithm for uniform measures is provably unbiased: we are guaranteed to obtain truly uniform samples from the suggestion database. The algorithm for popularity-induced distributions has some bias incurred by the fact suggestion services do not provide suggestion popularities explicitly.

The performance of suggestion sampling and mining is measured in terms of the number of queries sent to the suggestion server. This number is usually the main performance bottleneck, because queries require time-consuming communication over the Internet.

Using the publicly released AOL query log [22], we constructed a home-grown suggestion service. We used this service to empirically evaluate our algorithms. We found that the algorithm for uniform target measures is indeed unbiased and efficient. The algorithm for popularity-induced distributions requires more queries, since it needs to also estimate the popularity of each sample using additional queries.

Finally, we used both algorithms to mine the live suggestion services of two major search engines. We measured their database sizes and their overlap; the fraction of suggestions covered by Wikipedia articles; and the density of "dead links" in the search engines' indices when weighting pages by their ImpressionRank.

Techniques. Sampling suggestions directly from the target distribution is usually impossible, due to the restricted interface of the suggestion server. We therefore resort to the Monte Carlo simulation framework [18], used also in recent works on search engine sampling and measurements [3, 6, 4]. Rather than sampling directly from the target distribution, our algorithms generate samples from a different, easy-to-sample-from, distribution called the *trial distribution*. By weighting these samples using "importance weights" (specifying for each sample the ratio between its probabilities under the target and trial distributions), we *simulate* samples from the target distribution. As long as the trial and target distributions are sufficiently close, few samples from the trial distribution are need to simulate each sample from the target distribution.

When designing our algorithms we faced two technical challenges. The main challenge was to calculate importance weights in the case of popularity-induced target distributions. Note that suggestion services do not provide any explicit popularity information, so it may not be clear how to estimate the popularity-induced probability of a given suggestion. To overcome this difficulty we exploited the implicit scoring information latent in the rankings provided by the suggestion service as well as the power law distribution of query popularities learned from the AOL query log.

The second challenge was to create an efficient sampler that produces samples from a trial distribution that is close to the target distribution. While the correctness of our sampling algorithms is independent of the choice of the trial distribution (due to the use of Monte Carlo simulation), the efficiency of the algorithms (how many queries are needed to generate each sample) depends on the distance between

[6]Google Trends (`google.com/trends`) provides a similar functionality for popular queries, building on its privileged access to Google's query logs. Our techniques can be used with any search engine and with any keyword (regardless of its popularity), and without requiring privileged access to the log.

the trial and target distributions. We observe that a "flat sampler" (one that samples a random string and outputs the first suggestion succeeding this string in lexicographical order), which is popular in database sampling applications [24], is highly ineffective in our setting, as suggestions are not evenly spread in the string space. Empirical analysis of this sampler on the AOL data set revealed that about 10^{93}(!) samples from this sampler are needed to generate a single uniform sample.

We use instead a random walk sampler that can effectively "zero-in" on the suggestions (a similar sampler was proposed in a different context by Jerrum et al. [13]). This sampler relies on the availability of a black-box procedure that can estimate the weights of a given string x and all continuations thereof under the target distribution. We present an efficient heuristic implementation of this procedure, both for the uniform and the popularity-induced target distributions. In practice, these heuristics proved to be very effective. We note that in any case, the quality of the heuristics impacts only the efficiency the importance sampling estimator and not its accuracy.

2. RELATED WORK

Several previous works [5, 16, 11, 7, 3, 6, 4] studied methods for sampling random documents from a search engine's index using its public interface. The problem we study in this paper can be viewed as a special case of search engine sampling, because a suggestion service is a search engine whose "documents" are suggestions and whose "queries" are prefix strings. Most of the existing search engine sampling techniques rely on the availability of a large pool of queries that "covers" most of the documents indexed by the search engine. That is, they assume that every document belongs to the results of at least one query from the pool. In the suggestions setting, coming up with such a pool is very difficult. Since the suggestion server returns up to 10 completions for each prefix string, most suggestions are returned as a result of a small number of prefix strings. Thus, a pool that covers most suggestions must be almost of the same size as the suggestion database itself. This is of course impractical.

On the other hand, suggestion sampling has two unique characteristics that are used by the algorithms we consider in this paper. First, "queries" (= prefix strings) are always prefixes of "documents" (= suggestions). Second, the scoring function used in suggestion services is usually query-independent.

Jerrum et al. [13] presented an algorithm for uniform sampling from a large set of combinatorial structures, which is similar to our tree-based random walk sampler. However, since their work is mostly theoretical, they have not addressed some practical issues like how to estimate sub-tree volumes (see Section 6). Furthermore, they employed only the rather wasteful rejection sampling technique rather than the more efficient importance sampling.

Recently, Dasgupta, Das, and Mannila [9] presented a tree random walk algorithm for sampling records from a database that is hidden behind a web form. Here too, suggestion sampling can be cast as a special case of this problem, because suggestions represent database tuples, and each character position represents one attribute. Like Jerrum et al, Dasgupta et al. do not employ volume estimations or importance sampling. Another difference is that a key ingredient in their algorithm is a step in which the attributes are shuffled randomly and then queried one by one. In our setting, the server interface enables querying character positions only in order, making this technique inapplicable.

Several papers studied techniques for random sampling from B-trees [24, 21, 20]. Most of these assume the tree is balanced, and are therefore not efficient for highly unbalanced trees, as is the case with the suggestion TRIE. Moreover, these works assume data is stored only at leaves of the tree (in our case internal nodes of the TRIE can also be suggestions), that nodes at the same depth are connected by a linked list (does not hold in our setting), and that nodes store aggregate information about their descendants (again, not true in out setting).

Finally, another related problem is tree size estimation [15] by sampling nodes from the tree. The standard solutions for this problem employ importance sampling estimation. We use this estimator in our algorithm for sampling from popularity-induced distributions.

3. PRELIMINARIES

3.1 Sampling and mining suggestions

Let $\mathcal{S}$ be a *suggestion database*, where each suggestion is a string of length at most $\ell_{\max}$ over an alphabet Σ. The suggestion database is typically constructed by extracting terms or phrases from an underlying *base data set*, such as a document corpus, a query log, or a dictionary. The search engine defines a *scoring function*, $\mathit{score} : \mathcal{S} \to [0, \infty)$ over the suggestions in $\mathcal{S}$. The score of a suggestion x represents its prominence in the base data set. For example, if the base data set is a query log, the popularity of x in the log could be its score.[7] A *suggestion server* suggests auto-completions from $\mathcal{S}$ to given query strings. Formally, for a query string x, let $\mathsf{completions}(x) = \{z \in \mathcal{S} \mid x \text{ is a prefix of z}\}$. Given x, the suggestion server orders the suggestions in $\mathsf{completions}(x)$ using the scoring function s and returns the k top scoring ones. (k is a parameter of the suggestion server. $k = 10$ is a typical choice.)

Several remarks are in order. (1) Actual suggestion servers may return also suggestions that are not simple continuations of the query string x (e.g., suggest "George Bush" for "Bus" or "Britney Spears" for "Britni"). In this work we ignore such suggestions and focus only on simple string completions. (2) We assume x is a completion of itself. Thus, if $x \in \mathcal{S}$, then $x \in \mathsf{completions}(x)$. (3) The scoring function used by the suggestion server is query-independent. (4) The scoring function induces a partial order on the suggestion database. The partial order is typically extended into a total order, in order to break ties (this is necessary when more than k results compete for the top k slots). (5) In principle, there could be suggestions in $\mathcal{S}$ that are not returned by the suggestion server for any query string, because they are always beaten by other suggestions with higher score. Since there is no way to know about such suggestions externally, we do not view these as real suggestions, and eliminate them from the set $\mathcal{S}$.

In this paper we address the suggestion sampling and mining problems. In *suggestion sampling* there is a *target distribution* π on $\mathcal{S}$ and our goal is to generate independent

[7]Any scoring function is acceptable. For example, if the scoring is personalized, the resulting samples/estimates will reflect the scoring effectively "visible" to the user performing the experiment.

samples from π. In suggestion mining, we focus on mining tasks that can be expressed as computing discrete integrals over the suggestion database. Given a *target measure* $\pi : \mathcal{S} \to [0, \infty)$ and a *target function* $f : \mathcal{S} \to \mathbb{R}$, the integral of f relative to π is defined as:

$$\mathrm{Int}_\pi(f) = \sum_{x \in \mathcal{S}} f(x)\pi(x).$$

Many mining tasks can be written in this form. For example, the size of the suggestion database is the integral of the constant 1 function ($f(x) = 1, \forall x$) under the uniform measure $\pi(x) = 1, \forall x$. The average length of suggestions is the integral of the function $f(x) = |x|$ under the uniform probability measure $\pi(x) = 1/|\mathcal{S}|, \forall x$. For simplicity of exposition, we focus on scalar target functions, yet vector-valued functions can be handled using similar techniques.

We would like to design sampling and mining algorithms that access the suggestion database only through the public suggestion server interface. Such algorithms do not have any privileged access to the suggestion database or to the base data set. The prior knowledge these algorithms have about $\mathcal{S}$ is typically limited to the alphabet Σ and the maximum suggestion length $\ell_{\max}$, so even the size of the suggestion database $|\mathcal{S}|$ is not assumed to be known in advance. (In cases we assume more prior knowledge about $\mathcal{S}$, we will state this explicitly.) Due to bandwidth limits, the sampling and mining algorithms are constrained to submitting a small number of queries to the suggestion server. Thus, the number of requests sent to the server will be our main measure of cost for suggestion sampling/mining algorithms. The cost of computing the target function f on given instances is assumed to be negligible.

This conference version of the paper is devoted primarily to the suggestion mining problem. Similar techniques can be used to perform suggestion sampling.

3.2 Tree mining

We next show that suggestion mining can be expressed as a special case of *tree mining*. This will allow us to describe our algorithms using standard tree and graph terminology.

Throughout, we will use the following handy notation: given a tree T and a node $x \in T$, T_x is the subtree of T rooted at x.

Let T be a rooted tree of depth d. Suppose that some of the nodes of the tree are "marked" and the rest are "unmarked". We denote the set of marked nodes by $\mathcal{S}$.[8] Each marked node x has a non-negative *score*: *score*(x). We focus on tree mining tasks that can be expressed as computing an integral $\mathrm{Int}_\pi(f)$ of a target function $f : \mathcal{S} \to \mathbb{R}$ relative to a target measure $\pi : \mathcal{S} \to [0, \infty)$. The mining algorithm knows the tree T in advance but does not know a priori which of its nodes are marked and which are not. In order to gain information about the marked nodes, the algorithm can probe a *tree server*. Given a node $x \in T$, the tree server returns an ordered set $\mathsf{results}(x)$ — the k top scoring *marked* nodes in the subtree T_x. The goal is to design tree mining algorithms that send a minimal number of requests to the tree server.

The connection between suggestion mining and tree mining should be quite obvious. The tree T in the suggestion mining case is a TRIE representing all the strings of length at most $\ell_{\max}$ over Σ. Thus, T is a complete $|\Sigma|$-ary tree of depth $\ell_{\max}$. There is a natural 1-1 correspondence between the nodes of this tree and strings of length at most $\ell_{\max}$ over Σ. The nodes of T that correspond to suggestions are marked and the rest of the nodes are unmarked. Since the nodes in the subtree T_x correspond to all the possible continuations of the string x, the suggestion server returns exactly the top k marked nodes in T_x. We conclude that suggestion mining is a special case of tree mining, and thus from now on we primarily focus on tree mining (in cases where we use the specific properties of suggestion mining, this will be stated explicitly).

[8]We later show that in the case of suggestion mining, the marked nodes coincide with the suggestions, so both are denoted by $\mathcal{S}$.

3.3 Randomized mining

Due to the limited access to the set of marked nodes $\mathcal{S}$, it is typically infeasible for tree mining algorithms to compute the target integral *exactly* and *deterministically*. We therefore focus on *randomized* (Monte Carlo) algorithms that *estimate* the target integral. These estimations are based on random samples drawn from $\mathcal{S}$.

Our two measures of quality for randomized mining algorithms will be: (1) *bias*: the expected difference between the output of the algorithm and the target integral; and (2) the *variance* of the estimator. We will shoot for algorithms that have little bias and small variance. This will guarantee (via Chebyshev's inequality) that the algorithm produces a good approximation of the target integral with high probability.

3.4 Training and empirical evaluation

In order to train and evaluate the different mining algorithms we present in this paper, we built a suggestion server based on the AOL query logs [22]. For this data set we have "ground truth", and we can therefore use it to tune the parameters of our algorithms as well as to evaluate their quality and performance empirically.

The AOL query log consists of about 20M web queries (of which about 10M are unique), collected from about 650K users between March 2006 and May 2006. The data is sorted by anonymized user IDs and contains, in addition to the query strings, also the query submission times, the ranks of the results on which users clicked, and the landing URLs of the clicked results. We extracted 10M unique query strings and their popularities and discarded all other data.

From the unique query set we built a simple suggestion server, while filtering out a negligible amount of queries containing non-ASCII characters.

REMARK. In spite of the controversy surrounding the release of the AOL query log, we have chosen to use it in our experiments and not a proprietary private query log, for several reasons: (1) The prominent feature of our algorithms is that they do not need privileged access to search engine data; hence, we wanted to use a publicly available query log. (2) Among the public query logs, AOL's is the most recent and comprehensive. (3) The use of a public query log makes our results and experiments reproducible by anyone.

3.5 Importance sampling

A naive Monte Carlo algorithm for estimating the integral $\mathrm{Int}_\pi(f)$ draws random samples $X_1, \ldots, X_n$ from the target measure π and outputs $\frac{1}{n}\sum_{i=1}^{n} f(X_i)$. However, implementing such an estimator is usually infeasible in our setting, because: (1) π is not necessarily a proper distribution (i.e.,

$\sum_{x \in \mathcal{S}} \pi(x)$ need not equal 1); and (2) even if π is a distribution, we may not be able to sample from it directly using the restricted public interface of the tree server.

The estimators we consider in this paper use the *Importance Sampling* framework [19, 12]. Rather than sampling marked nodes from the target measure π, the algorithm draws n samples $X_1, \ldots, X_n$ from a different, easy-to-sample-from, *trial distribution* p. The sampler from p is required to provide with each sample X also the probability $p(X)$ with which this sample has been selected. The algorithm then calculates for each sample node X an *importance weight* $w(X) = \pi(X)/p(X)$. Finally, the algorithm outputs

$$\frac{1}{n}\sum_{i=1}^{n} w(X_i) f(X_i)$$

as its final estimate. It is easy to check that as long as the support of p is a superset of the support of π, this is indeed an unbiased estimator of $\mathrm{Int}_\pi(f)$.

In some cases, we will be able to compute the target measure only up to normalization. For example, when the target measure is the uniform distribution on $\mathcal{S}$ ($\pi(x) = 1/|\mathcal{S}|, \forall x$), we will be able to compute its unnormalized form only ($\hat{\pi}(x) = 1, \forall x$). If the estimator uses unnormalized importance weights instead of the real importance weights (i.e., $\hat{w}(X) = \hat{\pi}(X)/p(X)$), then the resulting estimate is far off the target integral by the unknown normalization factor. It turns out that $\frac{1}{n}\sum_{i=1}^{n} \hat{w}(X_i)$ is an unbiased estimator of the normalization factor, so we use the following *ratio importance sampling* estimator to estimate the target integral:

$$\frac{\sum_{i=1}^{n} \hat{w}(X_i) f(X_i)}{\sum_{i=1}^{n} \hat{w}(X_i)}.$$

It can be shown (cf. [18]) that this estimator has small bias, yet the bias diminishes to 0 as $n \to \infty$.

Also the variance of the importance sampling estimator diminishes to 0 as $n \to \infty$ [18] (this holds for both the regular estimator and the ratio estimator). This variance depends also on the variance of the basic estimator $w(X)f(X) = \frac{\pi(X)f(X)}{p(X)}$, where X is a random variable with distribution p. When the target function f is uncorrelated with the trial and target distributions, this variance depends primarily on the similarity between $p(X)$ and $\pi(X)$. The closer they are, the lower is the variance of the basic estimator, and thus the lower is the number of samples (n) needed in order to guarantee the importance sampling estimator has low variance. Liu's "rule of thumb" [17] quantifies this intuition: in order for the importance sampling estimator to achieve the same estimation variance as if estimating $\mathrm{Int}_\pi(f)$ by sampling m independent samples directly from π (or the normalized version of π, if π is not a proper distribution), $n = m(1 + \mathrm{var}(w(X)))$ samples from p are needed. We will therefore always strive to find a trial distribution for which $\mathrm{var}(w(X))$ is as small as possible.

In order to design a tree mining algorithm that uses the importance sampling framework, we need to accomplish two tasks: (1) build an efficient procedure for computing the target weight of each given node, at least up to normalization; (2) come up with a trial distribution p that has the following characteristics: (a) we can sample from p efficiently using the tree server; (b) we can calculate $p(x)$ easily for each sampled node x; (c) the support of p contains the support of π; (d) the variance of $w(X)$ is low.

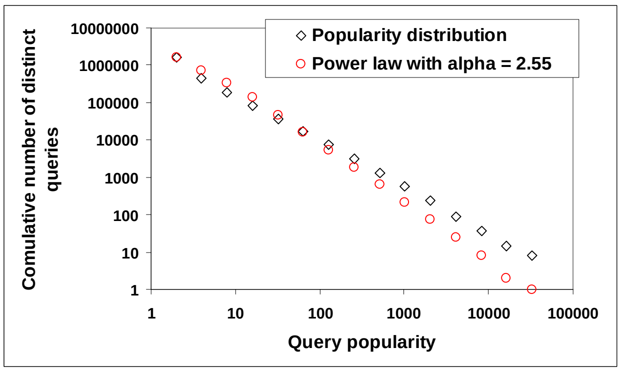


Figure 2: Cumulative number of distinct queries by popularity in the AOL data set (log-log scale). The graph indicates an almost power law with exponent $\alpha = 2.55$.

4. TARGET WEIGHT COMPUTATION

When the target measure is uniform ($\pi(x) = u(x) \triangleq 1, \forall x$), then computing the target weight is trivial. When the target measure is the uniform distribution ($\pi(x) = 1/|\mathcal{S}|, \forall x$), we compute the target weight only up to normalization, $\hat{\pi}(x) = u(x) = 1, \forall x$, and then apply ratio importance sampling.

The case the target measure is the score distribution ($\pi(x) = s(x) \triangleq \mathit{score}(x)/\mathit{score}(\mathcal{S})$) is much harder to deal with, as we do not receive any explicit score information from the tree server. Nevertheless, the server provides some cues about the scores of nodes, because it orders the results by their score. We will use these cues as well as the assumption that the scores have a power law distribution with a known parameter to sample nodes proportionally to their scores.

The rest of this section is devoted to describing our algorithm for estimating the target probability $s(x)$ (up to normalization) for the case π is the score distribution. We assume in this section that scores are integers, as is the case with popularity-based scores. This assumption only makes the algorithm more difficult to design (dealing with continuous scores is easier).

Score rank. Our first step is to reduce the problem of calculating $s(x)$ to the problem of calculating the relative "score rank" of x. Let us define the relative *score rank* of a node $x \in \mathcal{S}$ to be the fraction of nodes whose score is at least as high as the score of x:

$$\mathrm{srank}(x) = \frac{1}{|\mathcal{S}|}|\{y \in \mathcal{S} \mid \mathit{score}(y) \geq \mathit{score}(x)\}|.$$

Thus, $\mathrm{srank}(x) \in [1/|\mathcal{S}|, 1]$; it is at least $1/|\mathcal{S}|$ for the top scoring node(s) and it is 1 for the lowest scoring node(s).

It has been observed before (see, e.g., [23, 25]) that the popularity distribution of queries in query logs is close to being a power law. We verified this fact also on the AOL query log (see Figure 2). We found that the popularity distribution is at total variation distance distance of merely **11%** from a power law whose exponent is $\alpha = \mathbf{2.55}$

We can thus write the probability of a uniformly chosen node $X \in \mathcal{S}$ to have score at least s to be:

$$\Pr(\mathit{score}(X) \geq s) \approx \left(\frac{s_{\min}}{s}\right)^{\alpha - 1}, \qquad (1)$$

where $s_{\min}$ is the score of the lowest scoring node.

REMARK. The above expression for the cumulative density is only approximate, due to the following factors: (1) The score distribution is slightly different from a true power law. (2) The CDF of a discrete power law is approximated by the CDF of a continuous power law (see [8]). (3) The CDF of a "chopped" power law (all values above some finite n have probability 0) is approximated by the CDF of an unbounded power law. The latter two errors should not be significant when n is sufficiently large. For simplicity of exposition, we suppress the approximation error for the rest of the presentation. In Section 7, we provide empirical analysis of the overall estimation error incurred by our algorithm.

Note that for a given node x, $\Pr(score(X) \geq score(x)) = \text{srank}(x)$. Therefore, we can rewrite $score(x)$ in terms of $\text{srank}(x)$ as follows:

$$score(x) = \left[\frac{s_{\min}}{\text{srank}(x)^{1/(\alpha-1)}}\right]. \qquad (2)$$

(In order to make sure $score(x)$ is an integer, we round the fraction to the nearest integer (cf. [8]).) If both the exponent α and the minimum score $s_{\min}$ are known in advance, then by the above equation we can use $\text{srank}(x)$ to calculate $s(x) = score(x)/score(\mathcal{S})$ up to normalization: $\hat{s}(x) = score(x)$. The unknown normalization constant is $score(\mathcal{S})$ (recall that calculating $s(x)$ up to normalization is enough, because we can apply ratio importance sampling).

Typically, we expect the minimum score $s_{\min}$ to be 1. If we do not want to rely on the knowledge of $s_{\min}$, then we can set the unnormalized target weight as follows:

$$\hat{s}(x) = \frac{1}{\text{srank}(x)^{1/(\alpha-1)}}. \qquad (3)$$

Suppressing the small rounding error, $\hat{s}(x)$ is the same as $s(x)$, up to the unknown normalization constant $score(\mathcal{S})/s_{\min}$.

Why can we assume the exponent α to be known in advance? Due to the scale-free nature of power laws, the exponent α is independent of the size of the query log and of $s_{\min}$. α is dependent mainly on the intrinsic characteristics of the query log, which are determined by the profile of the search engine's user population. In our experiments, we assumed the AOL exponent ($\alpha = 2.55$) holds for other search engines as well, as we expect query logs of different major search engines to have similar profiles.[9] Note that using an inaccurate exponent can bias our target weight estimations. Thus, one should always use the most accurate and up-to-date information she has about the exponent of the search engine being mined.

Score rank estimation. We showed that in order to calculate the unnormalized target weight $\hat{s}(x)$ of a given node x, it suffices to calculate $\text{srank}(x)$. We next show how we estimate the score rank of a node by further reducing it to estimation of an integral relative to a uniform target measure.

Let $\text{sorder}(x, y)$ be the *score order* indicator function:

$$\text{sorder}(x,y) = \begin{cases} 1, & \text{if } score(x) \geq score(y), \\ 0, & \text{if } score(x) < score(y). \end{cases}$$

[9]Previous studies (e.g., [1, 2]) also reported estimates of query log exponents. Not all these reports are consistent, possibly due to differences in the query log profiles studied.

For a given node x, let $\text{sorder}_x(y) = \text{sorder}(x, y)$. We can write $\text{srank}(x)$ as an integral of sorder_x relative to the uniform distribution:

$$\text{srank}(x) = \frac{1}{|\mathcal{S}|}\sum_{y \in \mathcal{S}} \text{sorder}_x(y).$$

Now, assuming we can compute sorder (and thus sorder_x in particular), the expression for $\text{srank}(x)$ is a discrete integral of a computable function over the set of marked nodes $\mathcal{S}$ relative to the uniform distribution. We can thus apply our tree mining algorithm with the uniform target distribution to estimate this integral.

Score order approximation. Following the above reductions, we are left to show how to compute the score order $\text{sorder}(x, y)$ for two given nodes x, y.

The naive way to compute the score order of x, y would be to find their scores and compare them. This is of course infeasible, because we do not know how to calculate scores. A slight variation of this strategy is the following. Suppose we could come up with a function $b : \mathcal{S} \to \mathbb{R}$ that satisfies two properties: (1) b is easily computable; (2) b is *order-consistent* with s (that is, for every $x, y \in \mathcal{S}$, $score(x) \geq score(y)$ if and only if $b(x) \geq b(y)$). Given such a function b, we can compute $\text{sorder}(x, y)$ by computing $b(x)$ and $b(y)$ and comparing them.

We next present a function b that is relatively easy to compute and has high order correlation with s.

We say that an ancestor z of x *exposes* x, if x is one of the top k marked descendants of z, i.e., $x \in \textsf{results}(z)$. The *highest exposing ancestor* of x, $a(x)$, is the highest ancestor of x that exposes it.

Let $pos(x)$ be the position of x in $\textsf{results}(a(x))$. By definition, $score(x) \geq score(y)$ for every $y \in \mathcal{S}_{a(x)}$, except maybe for the $pos(x) - 1$ nodes that are ranked above x in $\textsf{results}(a(x))$. Due to the scale-free nature of power laws, we expect the distribution of scores in each subtree of T to be power-law distributed with the same exponent α. This means that the larger is the number of nodes that are scored lower-or-equal to x in $\mathcal{S}_{a(x)}$, the higher is its score. This motivates us to define $b(x)$ to be the number of nodes whose score is lower-or-equal to $score(x)$ in $\mathcal{S}_{a(x)}$:

$$b(x) = |\mathcal{S}_{a(x)}| - pos(x) + 1.$$

The above intuition suggests that b is order-consistent with s. Empirical analysis on the AOL query log indeed demonstrates that b is order-consistent with s for high scoring nodes. It is not doing as well for low-scoring nodes. The reason for this is the following. By its definition, b imposes a total order on nodes that belong to the same subtree. This is perfectly okay for high scoring nodes, where it is very rare for two nodes to have exactly the same score. On the other hand, for low scoring nodes (e.g., nodes of score 1), this introduces an undesirable distortion. To circumvent this problem, we flatten the values of b for nodes that are placed close the leaves of the tree:

$$b(x) = \begin{cases} |\mathcal{S}_{a(x)}| - pos(x) + 1, & \text{if } |\mathcal{S}_{a(x)}| - pos(x) + 1 > C, \\ 1, & \text{if } |\mathcal{S}_{a(x)}| - pos(x) + 1 \leq C, \end{cases}$$

where C is some tunable threshold parameter (we used $C = 20$ in our experiments). We measured the Kendall tau correlation[10] between the rankings induced by s and b on the AOL

[10]Kendall tau is a measure of correlation between rankings,

query log and found it to be **0.8**. This suggests that while not perfectly order-consistent, b has high order-correlation with s.

How can we compute $b(x)$ efficiently? In order to find the ancestor $a(x)$, we perform a binary search on x's ancestors. As we shall see below, the random walk sampler that produces x queries all its ancestors, so this step does not require any additional queries to the tree server. The position $pos(x)$ of x in $\mathsf{results}(a(x))$ can be directly extracted from $\mathsf{results}(a(x))$.

To estimate the volume $|\mathcal{S}_{a(z)}|$, we observe that it can be written as an integral of the constant 1 function over $\mathcal{S}_{h(x)}$ relative to the uniform measure:

$$|\mathcal{S}_{a(x)}| = \sum_{y \in \mathcal{S}_{a(x)}} 1.$$

Therefore, applying again our tree miner with a uniform target measure on the subtree $T_{a(x)}$, we can estimate this integral accurately and efficiently (note that our tree miners can mine not only the whole tree T but also subtrees of T).

Recap. The algorithm for estimating the unnormalized target weight $\hat{s}(x)$ of a given node x works as follows:

1. Sample nodes $Y_1, \ldots, Y_m$ from a trial distribution p on T using the random walk sampler (see Section 5).

2. Compute estimators $B(x)$ for $b(x)$ and $B(Y_1), \ldots, B(Y_m)$ for $b(Y_1), \ldots, b(Y_m)$, respectively (see below).

3. For each i, estimate $\text{sorder}_x(Y_i)$ as follows:
$$SO_x(Y_i) = \begin{cases} 1, & \text{if } B(x) \geq B(Y_i), \\ 0, & \text{otherwise.} \end{cases}$$

4. Estimate $\text{srank}(x)$ as
$$SR(x) = \frac{\sum_{i=1}^{m} \frac{1}{p(Y_i)} SO_x(Y_i)}{\sum_{i=1}^{m} \frac{1}{p(Y_i)}}.$$

5. Output the estimator for $\hat{s}(x)$:
$$\left\lceil \frac{s_{\min}}{SR(x)^{1/(\alpha-1)}} \right\rceil.$$

In order to estimate the value $b(y)$, we perform the following steps:

1. Extract $a(y)$ and $pos(y)$ from the list of y's ancestors.

2. Compute an estimate $V(a(y))$ for $|\mathcal{S}_{a(y)}|$ using the random walk sampler applied to the subtree $T_{a(y)}$ with a uniform target measure and a constant 1 function.

3. If $V(a(y)) - pos(x) + 1 > C$, set $B(y) = V(a(y)) - pos(y) + 1$. Otherwise, set $B(y) = 1$.

Recycling samples. The algorithm, as described, is rather costly. It runs three importance sampling estimators nested within each other. The first estimator generates n samples $X_1, \ldots, X_n$, in order to estimate an integral relative to the score distribution. For each sample X_i, the algorithm estimates $\hat{s}(X_i)$. To this end, the algorithm generates m samples $Y_1, \ldots, Y_m$ that are used to estimate $\text{srank}(X_i)$. Finally, for each Y_j, the algorithm generates ℓ samples $Z_1, \ldots, Z_\ell$ that are used to estimate $|\mathcal{S}_{a(Y_j)}|$. m, n, ℓ are chosen to be sufficiently large, so the corresponding estimators have low variances. To conclude, each integral estimation invokes the random walk sampler $m \cdot n \cdot \ell$ times, where each invocation incurs several queries to the tree server (see Section 5).

In order to reduce this significant cost, we *recycle* samples. Rather than generating fresh samples $Y_1, \ldots, Y_m$ for each X_i, we use $X_1, \ldots, X_n$ themselves as substitutes. That is, the second level estimator uses $X_1, \ldots, X_n$ to estimate $\text{srank}(X_i)$.

This recycling step reduces the number of samples per integral estimation from $mn\ell$ to $n\ell$. It introduces some dependencies among the samples from s, yet our empirical results indicate that this does not have a significant effect on the quality of the estimation.

Analysis. The algorithm for estimating $\hat{s}(x)$ incurs bias due to several factors:

1. As mentioned above, the expression (1) for $\text{srank}(x) = \Pr(S(X) \geq \mathit{score}(x))$ is only approximate. This means that $\hat{s}(x)$, as defined in Equation (3), is not exactly proportional to $s(x)$.

2. In the estimation of $\hat{s}(x)$, we do not compute $\text{srank}(x)$ deterministically, but rather only estimate it. Since $\hat{s}(x)$ is not a linear function of $\text{srank}(x)$, then even if we have an unbiased estimator for $\text{srank}(x)$, the resulting estimator for $\hat{s}(x)$ is biased.

3. In the estimation of $\text{srank}(x)$, we do not compute sorder_x, but rather only approximate it using the function b. Thus, the estimator for $\text{srank}(x)$ is biased.

4. The function b is not computed deterministically, but is rather estimated, again contributing to possible bias in the estimation of $\text{srank}(x)$.

We measured empirically the bias incurred by the above factor (excluding the score rank estimation bias) on the AOL data set, and found the total variation distance between the true score distribution and the distribution of target weights calculated by the above algorithm to be merely 22.74%.

5. RANDOM WALK SAMPLER

In this section we describe the trial distribution sampler used in our tree miners. The sampler is based on a random walk on the tree. It gradually collects cues from the tree and uses them to guide the random walk towards concentrations of marked nodes. Whenever the random walk reaches a marked node and the sampler decides it is time to stop, the reached node is returned as a sample.

We first need to introduce some terminology and notation. Recall that the algorithm knows the whole tree T in advance, but has no a priori information about which of its nodes are marked. The only way to gain information about marked nodes is by probing the tree server.

We use T_x to denote the subtree of T rooted at x and $\mathcal{S}_x$ to denote the set of marked nodes in T_x: $\mathcal{S}_x = T_x \cap \mathcal{S}$. We call $\hat{\pi}(\mathcal{S}_x) = \sum_{y \in \mathcal{S}_x} \hat{\pi}(y)$ the *volume of $\mathcal{S}_x$ under $\hat{\pi}$*, where $\hat{\pi}$ is an unnormalized form of the target distribution

which assumes values in the interval [-1,1]. It is 1 for identical rankings and -1 for opposite rankings. We used a variant of Kendall tau, due to Fagin et al. [10], that can deal with rankings with ties.

π (e.g., for a uniform target measure, the volume of $\mathcal{S}_x$ is simply its size). We say that a node x is *empty* if and only if $\hat{\pi}(\mathcal{S}_x) = 0$. Assuming the support of π covers all the marked nodes (which is the case both for the uniform and the score-induced distributions), every marked node is non-empty.

The random walk sampler uses as a black box two procedures: a "target weight estimator" and a "volume estimator". Given a node x, the target weight estimator, $\text{TargetEst}_{\hat{\pi}}(x)$, and the volume estimator, $\text{VolEst}_{\hat{\pi}}(x)$, return estimates of $\hat{\pi}(x)$ and $\hat{\pi}(\mathcal{S}_x)$, respectively. We describe the implementation of these procedures for the uniform and for the score-induced target measures in the next section.

The goal of the random walk sampler is to generate samples whose distribution is as close as possible to π. As we shall see below, the closeness of the sample distribution to π depends on the quality of the target weight and volume estimations. In particular, if $\text{TargetEst}_{\hat{\pi}}(x)$ always returns $\hat{\pi}(x)$ and $\text{VolEst}_{\hat{\pi}}(x)$ always returns $\hat{\pi}(\mathcal{S}_x)$, then the sample distribution of the random walk sampler is exactly π.

The main idea of the random walk sampler (see Figure 3) is as follows. The sampler performs a random walk on the tree starting at the root. At each step, the random walk either stops (in which case the reached node is returned as a sample), or continues to one of the children of the current node.

When visiting a node x, the sampler determines its next step in such a way that the probability to eventually return any node $z \in \mathcal{S}_x$ is proportional to the probability of z under the target distribution restricted to $\mathcal{S}_x$. In order to achieve this, the sampler computes a volume estimate $\text{VolEst}_{\hat{\pi}}(y)$, for each child y of x. y is selected as the next step with probability $\text{VolEst}_{\hat{\pi}}(\mathcal{S}_y)/\text{VolEst}_{\hat{\pi}}(\mathcal{S}_x)$. If x itself is marked, the sampler computes an estimate $\text{TargetEst}_{\hat{\pi}}(x)$ of its target weight and stops the random walk, returning x, with probability $\text{TargetEst}_{\hat{\pi}}(x)/\text{VolEst}_{\hat{\pi}}(\mathcal{S}_x)$.

Testing whether a node x is marked (line 9) is simple: we query the tree server for x and determine that it is marked if and only if x itself is returned as one of the results. Finding the non-empty children of a node x (line 4) is done by querying the tree server for all the children of x and checking whether each child y is marked or has a marked descendant (i.e., $\mathsf{results}(y) \neq \emptyset$).

The following proposition shows that in the ideal case the the estimators $\text{TargetEst}_{\hat{\pi}}$ and $\text{VolEst}_{\hat{\pi}}$ are always correct, the sampling distribution of the random walk sampler is exactly π:

Proposition 5.1. *If for all x, $\text{TargetEst}_{\hat{\pi}}(x) = \hat{\pi}(x)$ and $\text{VolEst}_{\hat{\pi}}(x) = \hat{\pi}(\mathcal{S}_x)$, then the sampling distribution of the random walk sampler is π.*

Proof. Let x be a marked node. Let $x_1, \ldots, x_t$ be the sequence of nodes along the path from the root of T to x. In particular, $x_1 = \text{root}(T)$ and $x_t = x$. x is selected as the sample node if and only if for each $i = 1, \ldots, t-1$, the sampler selects x_{i+1} as the next node and for $i = t$ it decides to stop at x_t. The probability of the sampler to select x_{i+1} as the next node of the random walk, conditioned on the current node being x_i, is $\hat{\pi}(\mathcal{S}_{x_{i+1}})/\hat{\pi}(\mathcal{S}_{x_i})$. The probability of the sampler to stop at x_t is $\hat{\pi}(x)/\hat{\pi}(\mathcal{S}_{x_t})$. Therefore, the probability of outputting $x = x_t$ as a sample is:

$$\prod_{i=1}^{t-1} \frac{\hat{\pi}(\mathcal{S}_{x_{i+1}})}{\hat{\pi}(\mathcal{S}_{x_i})} \cdot \frac{\hat{\pi}(x)}{\hat{\pi}(\mathcal{S}_{x_t})} = \frac{\hat{\pi}(x)}{\hat{\pi}(\mathcal{S}_{x_1})} = \frac{\hat{\pi}(x)}{\hat{\pi}(\mathcal{S})} = \pi(x).$$

```
procedure RWSampler($T$)
 1: $x$ := root($T$)
 2: trial_prob := 1
 3: while true do
 4:    children := GetNonEmptyChildren($x$)
 5:    for each child $y \in$ children do
 6:       $V(\mathcal{S}_y)$ := VolEst$_{\hat{\pi}}(y)$
 7:    end for
 8:    $V(\mathcal{S}_x)$ := $\sum_{y \in \text{children}} V(\mathcal{S}_y)$
 9:    if IsMarked($x$) then
10:       $T(x)$ := TargetEst$_{\hat{\pi}}(x)$
11:       $V(\mathcal{S}_x)$ := $V(\mathcal{S}_x) + T(x)$
12:       if children = $\emptyset$ then
13:          return ($x$, trial_prob)
14:       end if
15:       stop_prob := $T(x)/V(\mathcal{S}_x)$
16:       toss a coin whose heads probability is stop_prob
17:       if coin is heads then
18:          trial_prob := trial_prob $\cdot$ stop_prob
19:          return ($x$, trial_prob)
20:       else
21:          trial_prob := trial_prob $\cdot$ (1 - stop_prob)
22:       end if
23:    end if
24:    select $y \in$ children with probability $V(\mathcal{S}_y)/V(\mathcal{S}_x)$
25:    $x$ := $y$
26:    trial_prob := trial_prob $\cdot$ $V(\mathcal{S}_y)/V(\mathcal{S}_x)$
27: end while
```

Figure 3: The random walk sampler.

□

When the estimators $\text{TargetEst}_{\hat{\pi}}$ and $\text{VolEst}_{\hat{\pi}}$ are not accurate, the sampling distribution of the random walk sampler is no longer guaranteed to be exactly π. However, the better the estimations, the closer will be the sampling distribution to π. Recall that the distance of the sampling distribution of the random walk sampler from the target distribution π affects only the efficiency of the importance sampling estimator and not its accuracy.

6. VOLUME AND TARGET ESTIMATIONS

In this section we describe implementations of the estimators $\text{TargetEst}_{\hat{\pi}}$ and $\text{VolEst}_{\hat{\pi}}$. The implementations vary between the case $\hat{\pi}$ is a uniform measure u and the case π is a score-induced distribution s. In both cases we ensure that the support of the trial distribution induced by $\text{TargetEst}_{\hat{\pi}}$ and $\text{VolEst}_{\hat{\pi}}$ contains the support of the corresponding target measure.

The implementations were designed to be extremely cheap, requiring a small number of queries to the tree server per estimation. This design choice necessitated us to resort to super-fast heuristic estimations, which turned out to work well in practice. The savings gained by these cheap estimator implementations well paid off the reduced efficiency incurred by the less qualitative trial distributions.

6.1 Uniform target measure

Target weight estimation. When $\hat{\pi}$ is a uniform measure u, we set $\text{TargetEst}_u(x) = 1$ for every x, and thus the target weight estimator is trivial in this case.

Uniform volume estimation. Since $\hat{\pi} = u$ is the constant 1 function, $\text{VolEst}_u(\mathcal{S}_x) = |\mathcal{S}_x|$. We first show an accurate, yet inefficient, "naive volume calculator", and then propose a more efficient implementation.

Naive volume calculator. The naive volume calculator builds on the following observation: Whenever we send a node x, for which $|\mathcal{S}_x| < k$, to the tree server, the server returns *all* the marked descendants of x, and thus $|\mathcal{S}_x| = |\mathsf{results}(x)|$. Conversely, whenever $|\mathsf{results}(x)| < k$, we are guaranteed that $|\mathcal{S}_x| = |\mathsf{results}(x)|$. Therefore, the naive calculator applies the following recursion: it sends x to the tree server; if $|\mathsf{results}(x)| < k$, the calculator returns $|\mathsf{results}(x)|$. Otherwise, it recursively computes the volumes of the children of x and sums them up. It also adds 1 to the volume, if x itself is marked.

It is immediate to verify that this algorithm always returns the exact volume. Its main caveat is that it is very expensive. The number of queries needed to be sent to the server may be up to linear in the size of the subtree T_x. This could be prohibitive, especially for nodes that are close to the root of the tree.

The actual volume estimator, $\mathrm{VolEst}_\mathrm{u}$, applies three different volume estimators and then combines their outputs into a single estimation. We now overview each of the estimators and then describe the combining technique.

Naive estimator. The first volume estimator used by $\mathrm{VolEst}_\mathrm{u}$ is the naive estimator, which runs the naive calculator for *one* level, i.e., it sends only x to the tree server. It then outputs the following volume estimation:

$$\mathrm{VolEst}_\mathrm{u}^{\mathrm{naive}}(x) = \begin{cases} |\mathsf{results}(x)|, & \text{if } |\mathsf{results}(x)| < k, \\ a & \text{if } |\mathsf{results}(x)| = k. \end{cases}.$$

Thus, for each node x whose volume is $< k$, the estimator provides the exact volume. The estimator cannot differentiate among nodes whose volume is $\geq k$, and therefore outputs the same estimate a for all of them. Here $a \geq k$ is a tunable parameter of the estimator. $a = k$ is a conservative choice, meaning that $\mathrm{VolEst}_\mathrm{u}^{\mathrm{naive}}(x)$ is always a lower bound on $|\mathcal{S}_x|$. Higher values of a may give better estimations on average.

Sample-based estimator. The second estimator we consider assumes the sampler is given as prior knowledge a random sample $\mathcal{S}'$ of nodes from $\mathcal{S}$. Given such a random sample, we expect it to hit every subset of $\mathcal{S}$ at the right proportion. That is, for every subset $B \subseteq \mathcal{S}$, $\frac{|B \cap \mathcal{S}'|}{|\mathcal{S}'|} \approx \frac{|B|}{|\mathcal{S}|}$. In particular, $|B \cap \mathcal{S}'| \cdot \frac{|\mathcal{S}|}{|\mathcal{S}'|}$ is an unbiased estimator of $|B|$. Applying this for volume estimations, we obtain the following estimator for $|\mathcal{S}_x|$:

$$|\mathcal{S}'_x| \cdot \frac{|\mathcal{S}|}{|\mathcal{S}'|}.$$

Note that the more samples fall into $\mathcal{S}'_x$, the more accurate we expect this estimator to be. Therefore, we will use this estimator only as long as $|\mathcal{S}'_x| \geq b$, for some tunable parameter b.

In order to transform the above estimator into a practical volume estimator, we need to guess the ratio $|\mathcal{S}|/|\mathcal{S}'|$. Let c be another tunable parameter that represents the guess for $|\mathcal{S}|/|\mathcal{S}'|$. We use it to define the estimator as follows:

$$\mathrm{VolEst}_\mathrm{u}^{\mathrm{sample}}(x) = \begin{cases} c|\mathcal{S}'_x|, & \text{if } |\mathcal{S}'_x| \geq b, \\ 0, & \text{otherwise.} \end{cases}$$

We need to address two questions: (1) how can we get such a random sample of nodes from $\mathcal{S}$? (2) how can we guess the ratio $|\mathcal{S}|/|\mathcal{S}'|$ (which is equivalent to guessing the size of $\mathcal{S}$)?

Assuming query logs of different search engines have similar profiles, we can view a public query log (such as the AOL data set) as a random sample from hidden query logs. In reality, a public query log is not a truly random sample from hidden query logs, because: (1) query logs change over time, and thus the public query log may be outdated; (2) different search engines may have different user populations and therefore different query log profiles. Yet, as long as the public query log is sufficiently close to being representative of the hidden query log, the crude estimations produced from the public query log may be sufficient for our needs.

REMARK. We stress again that accurate volume estimation is needed only to obtain a better trial distribution and, consequently, a more efficient tree miner. If the AOL query log is not sufficiently representative of other query logs, as we speculate, then all this means is that our tree miner will be less efficient. Its accuracy, on the other hand, will not be affected, due to the use of importance sampling.

Guessing the ratio $|\mathcal{S}|/|\mathcal{S}'|$ is more tricky. Clearly, the further away c is from $|\mathcal{S}|/|\mathcal{S}'|$, the worse is the volume estimation, and consequently the trial distribution we obtain is further away from the target distribution. What we do in practice is simply "guess" c, run the importance sampling estimator once to estimate the size of $|\mathcal{S}|$ (possibly incurring many queries in this first run due to the error in c), and then use the estimated value of $|\mathcal{S}|$ to set a more reasonable value for c for further runs of the estimator.

Score-based estimator. The score-based estimator relies on the same observation we made in the computation of the function b in the algorithm for computing target weights (see Section 4): due to the scale-free nature of power laws, there is positive correlation between the score of a node x and the volume of $\mathcal{S}_{a(x)}$, where $a(x)$ is the highest ancestor of x, for which x is one of the top k scoring descendants. In Section 4, we used this observation to order scores using volumes. Here, we use it in the reverse direction: we estimate volumes using scores.

For a node x, let $score(\mathcal{S}_x)$ be the "score volume" of $\mathcal{S}_x$, i.e., $\sum_{z \in \mathcal{S}_x} score(z)$. Assuming the correlation between uniform volume and score volume, we have:

$$\frac{|\mathcal{S}_x|}{|\mathcal{S}_{\mathrm{parent}(x)}|} \approx \frac{score(\mathcal{S}_x)}{score(\mathcal{S}_{\mathrm{parent}(x)})}.$$

It follows that $|\mathcal{S}_x| \approx |\mathcal{S}_{\mathrm{parent}(x)}| \cdot (score(\mathcal{S}_x)/score(\mathcal{S}_{\mathrm{parent}(x)}))$. $|\mathcal{S}_x|$is then estimated by multiplying an estimate for $|\mathcal{S}_{\mathrm{parent}(x)}|$ with an estimate for $score(\mathcal{S}_x)/score(\mathcal{S}_{\mathrm{parent}(x)})$.

To estimate $|\mathcal{S}_{\mathrm{parent}(x)}|$, the score-based estimator uses the the naive and sample-based estimators:

$$v(\mathrm{parent}(x)) = \sum_{y \in \mathrm{child}(\mathrm{parent}(x))} \mathrm{VolEst}_\mathrm{u}^{\mathrm{naive}}(y) + \mathrm{VolEst}_\mathrm{u}^{\mathrm{sample}}(y).$$

Next, the score-based estimator looks for the node $z \in \mathcal{S}_x$ that ranks the highest in $\mathsf{results}(\mathrm{parent}(x))$ (if no such node exists, the score-based estimator outputs 0). Note that given the estimate for $|\mathcal{S}_{\mathrm{parent}(x)}|$, we can estimate the fraction of nodes in $\mathcal{S}_{\mathrm{parent}(x)}$ whose score is at least as high as $score(z)$ (the "score rank" of z): it is $pos(z)/v(\mathrm{parent}(x))$, where $pos(z)$ is the position of z in $\mathsf{results}(\mathrm{parent}(x))$.

Assuming the scores in $\mathcal{S}_{\mathrm{parent}(x)}$ have the same power law distribution as the global score distribution, we can use Equation 2 from Section 4 to estimate $score(z)$ from the score rank of z. Since $z \in \mathcal{S}_x$, $score(z)$ is a lower bound on $score(\mathcal{S}_x)$. The score-based estimator simply uses $score(z)$ as an estimate for $score(\mathcal{S}_x)$.

Finally, to estimate $score(\mathcal{S}_{\text{parent}(x)})$, the scored-based estimator uses the estimate of $|\mathcal{S}_{\text{parent}(z)}|$ as well as the assumed power law distribution to estimate the sum of the scores of all nodes in $\mathcal{S}_{\text{parent}(z)}$.

The combined estimator. The combined volume estimator for x is the sum of the naive, the sample-based, and the score-based estimators. Note that if $|\mathsf{results}(x)| < k$, the naive estimator produces the exact volume, so we set $\text{VolEst}_{\text{u}}(x) = \text{VolEst}_{\text{u}}^{\text{naive}}(x)$. Otherwise, if $|\mathsf{results}(x)| \geq k$,

$$\text{VolEst}_{\text{u}}(x) = \text{VolEst}_{\text{u}}^{\text{naive}}(x) + \text{VolEst}_{\text{u}}^{\text{sample}}(x) + \text{VolEst}_{\text{u}}^{\text{score}}(x).$$

The rationale for adding (and not, e.g., averaging) the estimations is that the accuracy of the sample-based and the score-based estimators is proportional to the magnitude of their output (the higher the values they output, the more likely they are to be accurate). The output of the naive estimator is always constrained to lie in the interval $[0, a]$. By adding the estimates, we give the score-based and sample-based estimators higher weight only when they are accurate. When they are inaccurate the naive estimator dominates the estimation.

We used the AOL query log to tune the different parameters of the above estimators as well as to evaluate their accuracy.

6.2 Score-induced target distribution

Target weight estimation. To implement $\text{TargetEst}_{\hat{\text{s}}}(x)$, we use the same algorithm we presented in Section 4 with one modification. We replace the costly uniform volume estimations applied at the computation of the function b with cheap uniform estimations VolEst_{u}, as described above.

Score volume estimation. To estimate the volume of $\mathcal{S}_x$ when π is the score distribution, we combine the target weight estimator described above with the uniform volume estimator. We simply set:

$$\text{VolEst}_{\hat{\text{s}}}(x) = \text{VolEst}_{\text{u}}(x) + \sum_{y \in \mathcal{S}_x} \text{TargetEst}_{\hat{\text{s}}}(y).$$

7. EXPERIMENTAL RESULTS

7.1 Bias estimation

We used the AOL query log to build a local suggestion service, which we could then use to empirically evaluate the accuracy and the performance of our algorithms. The score of each query was set to be the popularity of that query in the AOL query log.

We first ran the uniform sampler against the local suggestion service. We used the samples to estimate the suggestion database size. Figure 4 shows how the estimation bias approaches 0 as the estimator generates more samples (and thereby uses more queries). The empirical bias becomes essentially 0 after about 400K queries. However, it is already at most 3% after 50K queries.

In order to make sure the uniform and the score-induced samplers do not generate biased samples, we also studied the distribution of samples by their score. We ran the two samplers against the local suggestion service and collected 10,000 samples. Figure 5 shows the distribution of samples by their score. We ordered the queries in the data set by their score, from highest to lowest, and split them into 10 deciles. For the uniform sampler each decile corresponds to 10% of queries in the log, and for the score-induced sampler, each decile corresponds to 10% of the "score volume" (the sum of the scores of all queries). The results indicate that the uniform sampler is practically unbiased, while the score-induced sampler has minor bias towards medium-scored queries.

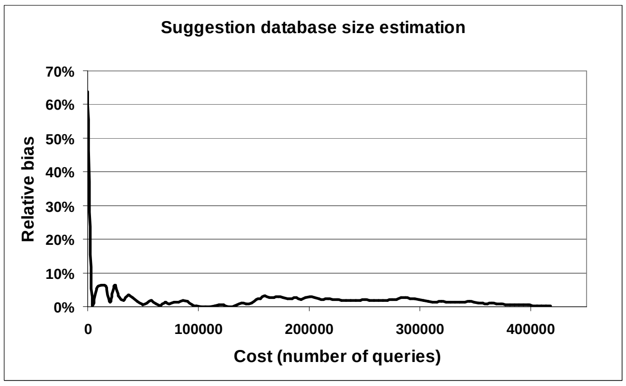


Figure 4: Relative bias versus query cost (uniform sampler, database size estimation).

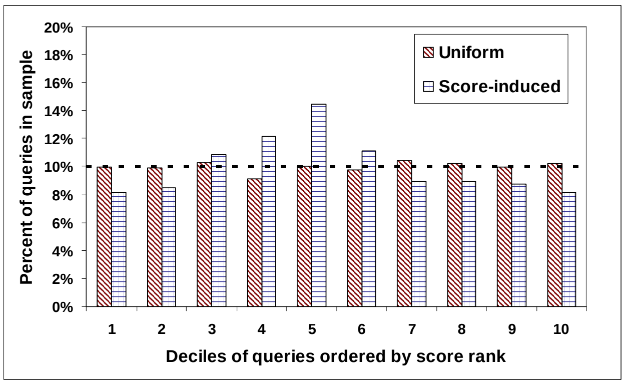
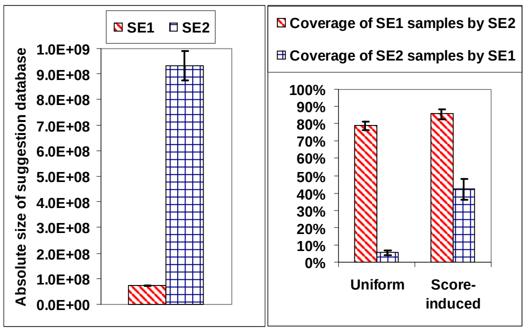


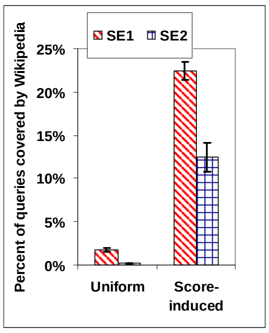
Figure 5: Distribution of samples by score.

7.2 Experiments on real suggest services

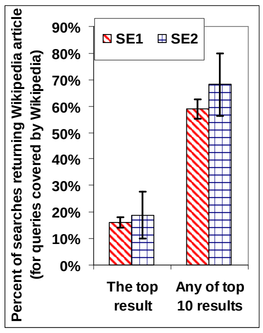
We ran the score-induced suggestion sampler on two live suggestion services, belonging to two large search engines. The experiments were performed in February-March 2008. The sampler used the AOL query log for the sample-based volume estimation. The sampler focused only on the standard ASCII alphanumeric alphabet, and thus practically ignored non-English suggestions. All suggestions were converted to lowercase. We sampled from a score-induced trial distribution, and, for each sample, computed both the uniform and the score-induced target weights. We thus could reuse the same samples for estimating functions under both uniform and score-induced distributions. Overall, we submitted 5.7M queries to the first suggestion service and 2.1M queries to the second suggestion service, resulting in 10,000 samples and 3,500 samples, respectively. The number of submitted queries includes the queries used for sampling suggestions as well as the addition queries used for score-



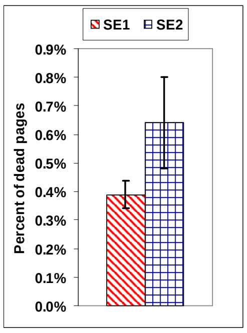
(a) Absolute and relative sizes of suggestion databases (English suggestions only).



(b) Suggestions covered by Wikipedia.



(c) Search results containing Wikipedia pages.



(d) Dead search results.

Figure 6: Experimental results on real suggest services.

induced target weight estimation.

We used the collected samples to estimate several parameters of the suggestion databases. Additionally, we submitted the sampled suggested queries to the corresponding search engines and obtained a sample of documents from their index, distributed according to ImpressionRank (see Section 1). We then estimated several parameters on that document sample.

The error bars throughout the section indicate standard deviation.

Estimation efficiency. The variance of the importance weights for the uniform and the score-induced target distribution was 6.7 and 7.7 respectively in SE1, and 13.5 and 43.7 respectively in SE2. That is, by Liu's rule of thumb, our estimator is at most 8 to 45 times less efficient than an estimator that could sample *directly* from the target distribution. These results indicate that our volume estimation technique is quite effective, even when using a relatively old AOL query log.

Suggestion database sizes. Figure 6(a) shows the absolute and the relative sizes of suggestion databases. Rather surprisingly, there is a large, 10-fold, gap between the number of distinct queries in the two databases. However, when measured according to the score-induced distribution, the relative sizes are much closer. These results indicate high overlap between popular suggestions in both databases, while the SE2 database contains many more less-popular suggestions.

Suggestion database coverage by Wikipedia. In this experiment we estimated the percentage of suggested queries for which there exists a Wikipedia article exactly matching the query. For example, for the query "star trek 2" we checked whether the web page `http://en.wikipedia.org/wiki/Star_Trek_2` exists. Figure 6(b) shows the percentage of queries having the corresponding Wikipedia article. While only a small fraction of the uniformly sampled suggestions are covered by Wikipedia, the score-proportional samples are covered much better. If we assume suggestion scores are proportional to query popularities, we can conclude that for more than 10% of the queries submitted to SE1 and for more than 20% of the queries submitted to SE2, there is an answer in Wikipedia (although, it may not be necessarily satisfactory).

ImpressionRank sampling. Our next goal was to estimate search engine index metrics relative to the ImpressionRank measure. To sample documents from the index proportionally to their ImpressionRank, we sampled suggestions from each of the two search engines proportionally to their popularity (again, we assume suggestion scores correspond to popularities). We submitted each sample suggestion to the search engine from whose suggestion service we sampled it and downloaded the result page, consisting of the top 10 results. Next, we sampled one result from each result page. We did not sample results uniformly, as it is known that the higher results are ranked the higher is the attention they receive from users. To determine the result sampling distribution, we used the study of Joachims et al. [14], who measured the relative number of times a search result is viewed as a function of its rank.

In the first experiment we estimated the reliance of search engines on Wikipedia as an authoritative source of search results. For queries having a relevant article in Wikipedia (as defined above), we measured the fraction of these queries on which the search engine returns the corresponding Wikipedia article as (1) the top result, and (2) one of the top 10 results. Figure 6(c) indicates that on almost 20% of queries that have a relevant Wikipedia article, search engines return that article as a top result. For more than 60% of these queries, the Wikipedia article is returned among top 10 results. This indicates a rather high reliance of search engines on Wikipedia as a source.

We next estimated the fraction of "dead links" (pages returning a 4xx HTTP return code) under ImpressionRank. Figure 6(d) shows that for both engines, less than 1% of the sampled results were dead. This indicates search engines do a relatively good job at cleaning and refreshing popular results. We note that one search engine returns slightly more dead links than the other.

Query popularity estimation. In this experiment we compared the relative popularity of various related keywords in the two search engines. Such comparisons can be useful to advertisers for selecting the targeting keywords for their ads. The results are shown in Table 1. In each category, keywords are ordered from the most popular to the least popular.

8. CONCLUSIONS

In this paper we introduced the suggestion sampling and mining problem. We described two algorithms for sugges-

SE1	SE2
barack obama	hillary clinton
john mccain	barack obama
george bush	john mccain
hillary clinton	george bush
aol	google
microsoft	aol
google	microsoft
yahoo	yahoo
madonna	britney spears
christina aguilera	madonna
britney spears	christina aguilera

Table 1: Queries ranked by their popularity.

tion sampling/mining: one for uniform target measures and another for score-induced distributions. The algorithms employ importance sampling to simulate samples from the target distribution by samples from a trial distribution. We show both analytically and empirically that the uniform sampler is unbiased and efficient. The score-induced sampler has some bias and is slightly less efficient, but is sufficiently effective to produce useful measurements.

Several remarks are in order about the applications of our algorithms to mining query logs. We stress that our algorithms do not compromise user privacy because: (1) they use only publicly available data provided by search engines; (2) they produce only aggregate statistical information about the suggestion database, which cannot be traced to a particular user.

A possible criticism of our methods is that they reflect the suggestion database more than the underlying the query log. Indeed, if in the production of the suggestion database, the search engine employs aggressive filtering or does not rank suggestions based on their popularity, then the same distortions may be surfaced by our algorithms. We note, however, that while search engines are likely to apply simple processing and filtering on the raw data, they typically try to maintain the characteristics of the original user-generated content, since these characteristics are the ones that guarantee the success of the suggestion service. We therefore believe that in most cases the measurements made by our algorithms are faithful to the truth.

Generating each sample suggestion using our methods requires sending a few thousands of queries to the suggestion server. While this may seem a high number, in fact when submitting the queries sequentially at sufficiently long time intervals, the added load on the suggestion server is marginal.

9. REFERENCES

[1] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. 30th SIGIR*, pages 183–190, 2007.

[2] R. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65, 2003.

[3] Z. Bar-Yossef and M. Gurevich. Random sampling from a search engine's index. In *Proc. 15th WWW*, pages 367–376, 2006.

[4] Z. Bar-Yossef and M. Gurevich. Efficient search engine measurements. In *Proc. 16th WWW*, pages 401–410, 2007.

[5] K. Bharat and A. Broder. A technique for measuring the relative size and overlap of public Web search engines. In *Proc. 7th WWW*, pages 379–388, 1998.

[6] A. Broder, M. Fontoura, V. Josifovski, R. Kumar, R. Motwani, S. Nabar, R. Panigrahy, A. Tomkins, and Y. Xu. Estimating corpus size via queries. In *Proc. 15th CIKM*, pages 594–603, 2006.

[7] M. Cheney and M. Perry. A comparison of the size of the Yahoo! and Google indices. Available at `http://vburton.ncsa.uiuc.edu/indexsize.html`, 2005.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data, 2007.

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

[10] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee. Comparing and aggregating rankings with ties. In *Proc. 23rd PODS*, pages 47–58, 2004.

[11] A. Gulli and A. Signorini. The indexable Web is more than 11.5 billion pages. In *Proc. 14th WWW*, pages 902–903, 2005.

[12] T. C. Hesterberg. *Advances in Importance Sampling*. PhD thesis, Stanford University, 1988.

[13] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43(2-3):169–188, 1986.

[14] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In *Proc. 28th SIGIR*, pages 154–161, 2005.

[15] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[16] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 5360(280):98, 1998.

[17] J. S. Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6:113–119, 1996.

[18] J. S. Liu. *Monte Carlo Strategies in Scientific Computing.* Springer, 2001.

[19] A. W. Marshal. The use of multi-stage sampling schemes in Monte Carlo computations. In *Symposium on Monte Carlo Methods*, pages 123–140, 1956.

[20] F. Olken and D. Rotem. Random sampling from B+ trees. In *Proc. 15th VLDB*, pages 269–277, 1989.

[21] F. Olken and D. Rotem. Random sampling from databases - a survey. *Statistics & Computing*, 5(1):25–42, 1995.

[22] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proc. 1st InfoScale*, 2006.

[23] P. C. Saraiva, E. S. de Moura, R. C. Fonseca, W. M. Jr., B. A. Ribeiro-Neto, and N. Ziviani. Rank-preserving two-level caching for scalable search engines. In *Proc. 24th SIGIR*, pages 51–58, 2001.

[24] C. K. Wong and M. C. Easton. An efficient method for weighted sampling without replacement. *SIAM J. on Computing*, 9(1):111–113, 1980.

[25] Y. Xie and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proc. 21st INFOCOM*, 2002.