Hashed Samples: Selectivity Estimators For Set Similarity Selection Queries

Marios Hadjieleftheriou †, Xiaohui Yu ‡, Nick Koudas ♮, Divesh Srivastava †

† *AT&T Labs–Research*
Florham Park NJ, USA
{marioh, divesh}@research.att.com

‡ *York University*
Toronto ON, Canada
xhyu@yorku.ca

♮ *University of Toronto*
Toronto ON, Canada
koudas@cs.toronto.edu

ABSTRACT

We study selectivity estimation techniques for set similarity queries. A wide variety of similarity measures for sets have been proposed in the past. In this work we concentrate on the class of weighted similarity measures (e.g., `TF/IDF` and `BM25` cosine similarity and variants) and design selectivity estimators based on a priori constructed samples. First, we study the pitfalls associated with straightforward applications of random sampling, and argue that care needs to be taken in how the samples are constructed; uniform random sampling yields very low accuracy, while query sensitive real-time sampling is more expensive than exact solutions (both in CPU and I/O cost). We show how to build robust samples a priori, based on existing synopses for distinct value estimation. We prove the accuracy of our technique theoretically, and verify its performance experimentally. Our algorithm is orders of magnitude faster than exact solutions and has very small space overhead.

1. INTRODUCTION

Data collections often have inconsistencies that arise due to a variety of reasons, such as typographic mistakes, formatting conventions, data transformation errors and more. Consistent or clean data are of high monetary significance for business practices; it is desirable to be able to identify and resolve such inconsistencies efficiently. For that purpose, various *string similarity* operators have been proposed in the past [2, 4, 5, 13, 30]. The main idea behind such operators is to view operands as sets of tokens and evaluate the similarity of the operand sets. If the similarity is high enough the operand pair is flagged as being of interest (e.g., potential duplicate). Furthermore, set similarity operators can also be used to evaluate similarity of set-valued attributes in general (e.g., in an Object Relational DBMS). The bulk of algorithm development in this area has concentrated on the efficient execution of join and selection operations. Hence, it is of interest to be able to efficiently and accurately evaluate the selectivity of such similarity operations for the purpose of query optimization [25, 27]. In this work we concentrate on *selectivity estimation* for set similarity selection queries.

Let $\mathcal{I}$ be a predefined set similarity measure. Given a query q, the goal of set similarity selection queries is to identify all sets with score greater than some user defined threshold τ. The goal of selectivity estimation is to *estimate the number* of such sets in the database. Formally, given a query q and a collection of sets D, estimate the size of the answer set $\{s \in D | \mathcal{I}(q, s) \geq \tau\}$. A large number of similarity measures have been proposed in the past (Jaccard, edit distance, cosine similarity, etc.). It has been demonstrated that no single similarity function is best across all application domains [9, 29]. For our purposes we will concentrate only on well known and largely deployed weighted similarity measures (e.g., `TF/IDF` cosine similarity). Efficiently evaluating such similarity measures is accomplished by means of specialized inverted indexes on the distinct tokens contained in the input sets (inverted indexes are built either in the form of relational tables using existing DBMS technology or as auxiliary files stored on secondary storage [15, 30]). Our goal is to design selectivity estimation techniques that exploit these specialized inverted indexes to provide robust estimation at minimal computational cost and storage overhead.

A simplistic approach for performing selectivity estimation is to take a random sample of the input sets, evaluate the similarity of the given query with the sampled sets and scale up the result. A better approach would be to use a random sample taken only with respect to the input sets that contain at least one token in common with the query, since only those sets have similarity greater than zero. We show that the former has very low accuracy for arbitrary queries, while the latter is more expensive than exact solutions (both in terms of CPU and I/O cost). We propose a new selectivity estimation technique, based on existing synopses for distinct value estimation, that builds samples a priori but is nevertheless able to provide very accurate estimates, very fast, for arbitrary queries. In addition, the new estimator can be updated very efficiently, under arbitrary updates, in contrast with the straightforward alternatives.

Section 2 presents essential background on set similarity retrieval. Section 3 discusses the straightforward solutions. Section 4 presents our proposed solution. Section 5 compares the properties of these alternatives approaches. Section 6 presents a thorough experimental evaluation. Section 7 discusses related work. Section 8 concludes the paper.

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

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

2. BACKGROUND

In this section we provide some necessary background on TF/IDF similarity and specialized indexes [18] for answering TF/IDF based selection queries. For illustration purposes, consider strings decomposed into words as our operand sets. For example, let strings s_1 = "Main St., Main" and s_2 = "Main St., Maine" be mapped into token multi-sets {'Main', 'St.', 'Main'} and {'Main', 'St.', 'Maine'}. The two multi-sets share two tokens in common. Clearly, the larger the intersection of the two multi-sets, the larger the potential similarity. Tokens that appear very frequently in the database (like 'Main' or 'St.') carry small information content, whereas rare tokens (like 'Maine') are more important semantically. Hence, the more important a token is, the larger the role it should play in overall similarity. For that reason, weighted similarity measures (for example TF/IDF) use the Inverse Document Frequency (idf) as token weights. The idf of a token is the inverse of the total number of times that this token appears in the data collection. In addition, weighted measures also use a Term Frequency (tf) component, i.e., each token is also weighted with respect to the total number of times it appears in the multi-set. In the rest, we concentrate on TF/IDF for simplicity, but our discussions can be extended to other measures as well (for example BM25 [4]).

Formally, consider a collection of sets D (e.g., a collection of strings where each string has been decomposed into q-grams, words, etc.), where every set consists of a number of elements from universe $\mathcal{U}$; Let set $s = \{t_1, \dots, t_n\}, t_i \in \mathcal{U}$. Every t_i is assigned an idf weight computed as follows: Let $N(t_i)$ be the total number of sets containing token t_i and N be the total number of sets in D. Then:

$$idf(t_i) = \log_2 (1 + N/N(t_i)).$$

Denote the term frequency of token t_i in set s by $tf(t_i, s)$. The normalized length of set s is computed as:

$$len(s) = \sqrt{\sum_{t_i \in s} tf(t_i, s)^2 \cdot idf(t_i)^2}.$$

The length normalized TF/IDF similarity of sets q and s is:

$$\mathcal{I}(q, s) = \sum_{t_i \in q \cap s} \frac{tf(t_i, s) \cdot tf(t_i, q) \cdot idf(t_i)^2}{len(s) \cdot len(q)}. \quad (1)$$

Length normalization restricts similarity in the interval $[0, 1]$. If $q = s$, the TF/IDF score is equal to 1. Otherwise, as the number of common tokens grows the score becomes larger. Nevertheless, the contribution of every common token to the score is dampened as the length divergence between the two sets grows.

Typical set similarity selection algorithms evaluate queries using some form of inverted indexes built on the tokens in $\mathcal{U}$. It is easy to show that Equation (1) obeys monotonicity, which enables the evaluation of similarity queries using TA/NRA style algorithms using the inverted index [30, 18]. Denote with $w(t_i, s)$ the partial weight contribution of token $t_i \in s$, to $\mathcal{I}(q, s)$, for arbitrary q. That is (refer to Equation (1)):

$$w(t_i, s) = \frac{tf(t_i, s) \cdot idf(t_i)}{len(s)}. \quad (2)$$

Now, construct one inverted list per token $t_i \in \mathcal{U}$, that consists of one pair $\langle s, w(t_i, s)\rangle$ per set s containing t_i (see Figure 1). Denote the list corresponding to token t_i by $\vec{t_i}$. Let

t_1 id	t_1 w	t_2 id	t_2 w	t_3 id	t_3 w
1	.7	2	.4	3	.1
2	.5	3	.4	4	.1
4	.5	4	.4	7	.1
5	.1	6	.1		...
	...		...	8	.1
					...

Figure 1: Inverted lists sorted by decreasing token contribution in the overall score.

query $q = \{t_1, \dots, t_n\}$ and length $len(q)$. By directly scanning inverted lists $\vec{t_1}, \dots, \vec{t_n}$ we can compute $\mathcal{I}(q, s)$ for all s in one pass, and report the ones that exceed threshold τ. Notice that irrelevant sets (with $s \cap q = \varnothing$) are never accessed. Alternatively, assume that lists are sorted in decreasing $w(t_i, s)$ order. Given that TF/IDF is a monotonic score function, we can now use TA/NRA style algorithms to compute the scores incrementally, and potentially terminate before exhaustively reading the lists. For example, the NRA algorithm reads lists in a round-robin fashion and iteratively loads the next element from every list starting from the top (see Figure 1). It maintains an *in memory hash table* with one entry per set id read. Each entry contains the aggregated score of the contributions of the lists where this id has already appeared in. It also contains a bit vector indicating the lists where this id has not been encountered yet. As more set ids are read from the lists, scores are completed and the algorithm reports sets with similarity above the threshold. Special boundary properties enable early termination of the algorithm when it is deemed that no encountered or yet unseen candidates can exceed the threshold.

It is clear that computing the exact answer of a query incurs both an I/O cost for retrieving elements from the inverted lists, and the computational cost of keeping the in memory candidate set up-to-date. Straightforwardly, any selectivity estimation technique should be *orders of magnitude faster* than exact evaluation, decreasing both I/O and CPU costs, while at the same time consuming as little extra space as possible and providing accurate estimates.

3. STRAIGHTFORWARD SOLUTIONS

In this section we present some straightforward solutions for selectivity estimation of set similarity selection queries, and identify the pitfalls. First, we introduce some useful notation. Let query $q = \{t_1, \dots, t_n\}$. Let $q_\cup = \vec{t_1} \cup \dots \cup \vec{t_n}$, the multi-set union of set ids contained in the inverted lists (where every id might be associated with multiple partial weights). Denote with $|x|$ the number of elements in multi-set x, and $|x|_d$ the distinct number of elements in multi-set x.

3.1 A Priori Computed Samples

Consider a uniform random sample S, drawn from all $s \in D$. A selectivity estimate is computed as:

$$A = |A_S| \cdot \frac{|D|}{|S|}, \quad (3)$$

where $A_S = \{s \in S : \mathcal{I}(q, s) \geq \tau\}$ (i.e., the number of query answers contained in the sample). Notice that in order to compute the similarity score $\mathcal{I}(q, s)$ for all $s \in S$, we need to

store in the sample the actual sets instead of their associated ids, which increases the space budget of the sample significantly. The fact that the sample needs to contain the actual sets is a huge drawback in terms of the real space used versus the effective size of the sample. As the sample becomes larger (to increase estimation accuracy), query evaluation becomes more expensive and hence benefits over exact algorithms become slimmer. Random sampling provides standard guarantees on the error of the estimated frequencies (more details appear in Section 4.2).

Another approach is to independently draw and store one uniform random sample per inverted list. Let list $\vec{t_i}$ (containing $\langle s, w(t_i, s)\rangle$ pairs) and denote with $\tilde{t}_i$ a random sample drawn from $\vec{t_i}$. Given query q, we compute one estimate per sample $\tilde{t}_i$ and report as answer the median, max, average or any other robust estimator. The independent estimate from each sample is computed as:

$$A = |A_{\tilde{t}_i}| \cdot \frac{|\vec{t_i}|}{|\tilde{t}_i|}. \tag{4}$$

Here we assume again that the actual sets are stored in the samples, in order to efficiently compute the answer size $A_{\tilde{t}_i}$. Hence, the total size of the sampled lists is equivalent to the size of simple random sampling. Moreover, this approach assumes independence across lists and ignores important correlations that possibly exist between the tokens, hence it will not work better than simple random sampling in practice.

It would be tempting to compute the sample union $\tilde{q}_\cup = \tilde{t}_1 \cup \ldots \cup \tilde{t}_n$, given query q, of list samples to solve this problem, but this would produce a biased sample, since there might be duplicate set ids among the sampled lists. Eliminating duplicates does not help because it would necessitate the computation of quantity $|q_\cup|_d$ (the distinct number of set ids in the union of the query lists), in order to scale up the result. The cost of computing $|q_\cup|_d$ is prohibitive, since it is larger than the cost of running exact algorithms (e.g., TA/NRA). Also, this approach is not using the partial weight information contained in the lists, since it is computing exact scores by storing the actual sets in the samples.

Finally, building samples a priori introduces another important aspect to this problem — that of efficiently handling updates. Even though random samples can be maintained efficiently in the presence of insertions, maintenance in the presence of deletions is expensive (an adversarial sequence of updates can result in an empty sample). Ideally, maintaining the uniformity of the samples would require resampling a given list every time an element is deleted.

3.2 Dynamically Computed Samples

A promising approach for solving the problem would be to construct a query sensitive sample in real time from the lists in q only. In order to dynamically construct a sample without having to exhaustively read the inverted lists (which would outweigh the benefit of estimation), the obvious choice is to use reservoir sampling [33].

Consider the following straightforward algorithm. Predefine a reservoir size S (e.g., equal to 5% of $|q_\cup|$, which is the size of the set that the exact solutions are working with) and use reservoir sampling to dynamically build a uniform random sample from $q_\cup$. Reservoir sampling starts by sequentially reading elements from the first list. Once the reservoir is full, it skips over a group of set ids using random seeks that follow a geometric distribution of jumps (the algorithm reduces unnecessary processing of elements that would not be sampled in the first place, by skipping over elements of the input [33]). Then, it reads the next id and randomly evicts an entry from the reservoir in order to make space for the newly sampled id. The process continues until all lists have been exhausted. The problem with this algorithm is that since lists might contain duplicate ids, the resulting reservoir is not a uniform random sample. (Reservoir sampling can be used only for sampling without replacement.) Hence, producing an unbiased estimate from this sample is not possible, and no theoretical guarantees can be provided for the estimation accuracy of this technique.

Moreover, in order to scale up the result correctly, it is essential to know $|q_\cup|_d$ (the domain size). Exactly computing this quantity is impractical, since it is costlier than running exact algorithms. An alternative is to estimate this quantity. One could use the sample itself to estimate the distinct union size. Albeit, uniform sampling is a notoriously bad distinct value estimator (e.g., see [10]), and hence cannot be used productively. Another option is to maintain one distinct value estimation synopsis per list $\vec{t_i}$ (e.g., FM [14] or KMV [6] sketches). However, specialized synopses occupy space proportional to the sample in order to provide estimates of comparable accuracy, and hence incur a large space cost, which offsets one of the benefits of dynamically computing the samples.

Finally, computing the number of answers A_S contained in a sample created dynamically needs further attention. Since elements of the input have been skipped, any dynamically constructed sample is not guaranteed to contain all the partial weight information $w(t_i, s)$ per sampled set s needed for reconstructing score $\mathcal{I}(q, s)$. Straightforwardly, the only possible way to compute exact scores is to retrieve the actual sets or store the sets in the samples. Clearly, retrievals incur additional random I/Os, while storing the actual sets in the sample, reduces the effective sample size for a fixed reservoir S (we illustrate this point in the experimental evaluation in Section 6). Furthermore, Haas et al. [17] showed that the cost of sampling with random accesses is greater than that of a sequential scan of the data for sampling rates greater than 2% under certain assumptions.

To summarize, there are three pitfalls associated with dynamically constructed samples in this setting. First, since the scores of the elements cannot be reconstructed using the sample a large number of random accesses need to be performed to fetch actual sets. Alternatively, the actual sets need to be stored in the sample, which decreases the effective size of the sample. Second, additional random seeks for skipping over the input need to be performed. Third, estimating the distinct size of arbitrary list unions requires maintaining separate distinct value estimation synopses of considerable size.

Our focus in the rest of the paper will be to build a priori random samples that circumvent all the pitfalls encountered above. We will design a technique that:

- Builds uniform samples from arbitrary combinations of inverted lists.
- Eliminates the need to store the actual sets in the sample, and computes scores efficiently.
- Eliminates the need of maintaining special distinct value estimation synopses.

- Provides unbiased estimates with small variance, outperforming alternatives.
- Handles updates gracefully, at a minimal cost, which is a departure from existing work on selectivity estimation using sampling.

4. HASHED SAMPLES

4.1 Construction, Querying and Updating

Using one sample for all queries is not robust in terms of space budget used versus effective sample size, as shown in the previous section. Also, since the sample is not focused for the specific query at hand, estimates are expected to have large variance. On the other hand, constructing one sample per list allows us to combine arbitrary list samples into a more focused, query specific sample that is expected to yield results of much lower variance.

Assume that we are given a predetermined space budget S. Clearly, there are two important aspects in utilizing the available budget efficiently when constructing individual samples per list. First, we need to leverage the partial weights contained in the inverted lists correctly in order to avoid storing actual sets in the samples, in contrast to the simple random sampling method. Second, we need to avoid maintaining separate distinct value estimation synopses, since these will further limit the available budget.

The first observation suggests that drawing independent samples from every list is not a viable option. Instead, we need to guarantee that if a set id is sampled in one list, it will be consistently sampled in all other lists that it appears in. Let query $q = \{t_1, \ldots, t_n\}$, and let $\tilde{t}_1, \ldots, \tilde{t}_n$ be precomputed samples of inverted lists $\vec{t}_1, \ldots, \vec{t}_n$ (where each sample contains pairs $\langle s, w(t_i, s)\rangle$). We would like to guarantee that given the union of samples $\tilde{q}_\cup = \tilde{t}_1 \cup \ldots \cup \tilde{t}_n$ and an arbitrary set id $s \in \tilde{q}_\cup$, all relevant partial weights $w(t_i, s)$ for computing $\mathcal{I}(q, s)$ are already included in $\tilde{q}_\cup$. At the same time, it is important to guarantee that $\tilde{q}_\cup$ is a uniform random sample of the lists in q (for arbitrary q), in order to be able to provide unbiased estimates.

This leads to the following observation. We need a property stronger than choosing samples uniformly at random. We need a procedure that picks samples consistently. This can be accomplished as follows: Impose a random permutation on the domain of set ids $\mathcal{D}$, and choose from every list a consistent subset of the permuted ids. Choose a family of universal hash functions $\mathcal{H}$ and randomly pick a hash function $h : \mathcal{D} \rightarrow \mathcal{P}$ [8]. The values $h(s_1), \ldots, h(s_{|\mathcal{D}|})$ will appear to be a sequence of i.i.d. samples from the discrete uniform distribution over $\mathcal{D}$ [1]. The hash function, *in an empirical sense that suffices for applications*, imposes a random permutation of the elements in $\mathcal{D}$.

We create the random samples as follows. Randomly choose hash function $h \in \mathcal{H}$. For simplicity, let hash function h distribute values in the interval $[1, 100]$. Choose a value $x \in [1, 100]$, and sample from every list $\vec{t}_i \in \mathcal{U}$ all set ids s with hash value $h(s) \leq x$. Given that the hash function is distributing ids uniformly in interval $[1, 100]$, this approach will approximately result in an $x\%$ sample of list $\vec{t}_i$ and, naturally, in an $x\%$ overall sample of the inverted lists. Given the total size of the inverted lists and budget S, we can deduce a maximum value x easily, in order to meet our budget. We call this algorithm Hashed Sampling (HS).

Algorithm 1: CS (sample construction algorithm).

Input : Lists $t_1, \ldots, t_{|\mathcal{U}|}$, Hash function $h \in \mathcal{H} : \mathbb{N} \rightarrow [1, 100]$, Sample size $x\%$
Output: Sampled lists $\tilde{t}_1, \ldots, \tilde{t}_{|\mathcal{U}|}$
forall $1 \leq i \leq |\mathcal{U}|$ **do**
    forall $p = \langle s, w(t_i, s)\rangle \in \vec{t}_i$ *s.t.* $h(s) \leq x$ **do**
        Insert p in $\tilde{t}_i$
    end
end

	1	2	3	4	5			m
	s_1	s_2	s_3	s_4	s_5			s_m
t^1	1	1	1	0	0	$\cdots$	0	1
t^2	0	1	1	1	0	$\cdots$	0	0
$\vdots$			$\vdots$					$\vdots$
t^n	1	0	0	0	1	$\cdots$	1	0

$h(s_4) = 1, h(s_m) = 2, \ldots$

	s_4	s_m	s_1	s_2	s_9			s_3
t_1	0	1	1	1	1	$\cdots$	0	1
t_2	1	0	0	1	1	$\cdots$	1	1
$\vdots$			$\vdots$					$\vdots$
t_n	0	0	1	0	1	$\cdots$	1	0

Figure 2: Top: The original universe $\vec{t}_1 \cup \ldots \cup \vec{t}_n$. Bottom: A random permutation using hash function h. Any prefix of set ids in the permutation, is equivalent to a uniform random sample from $\vec{t}_1 \cup \ldots \cup \vec{t}_n$. The extra information within the columns of the matrix, provides only the necessary partial weights for computing the scores.

Denote with h_i the maximum hash value contained in $\tilde{t}_i$. Notice that a given sample $\tilde{t}_i$ might not necessarily contain a set id with hash value x. Nevertheless, it is guaranteed by construction that list $\vec{t}_i$ did not contain any id s s.t. $h_i < h(s) \leq x$, and hence we can safely assume that $h_i = x = h_m$ for all $\vec{t}_i$. Hence, we can construct the sample union $\tilde{q}_\cup$ directly to be the union of all sampled ids. The procedure appears as Algorithm 1.

We can estimate the selectivity of $q = \{t_1, \ldots, t_n\}$ as follows. Construct the sample union $\tilde{q}_\cup = \tilde{t}_1 \cup \ldots \cup \tilde{t}_n$. The estimate from the sample is:

$$A = |A_{\tilde{q}_\cup}| \cdot \frac{|q_\cup|_d}{|\tilde{q}_\cup|_d}. \tag{5}$$

By construction of the sampled lists, it is guaranteed that for every sampled id $s \in \tilde{q}_\cup$, all partial weights $w(t_i, s)$ in lists $\vec{t}_1, \ldots, \vec{t}_n$ are present in $\tilde{q}_\cup$, and thus the scores $\mathcal{I}(q, s)$ can be computed directly from the sample. In addition, due to the properties of the random permutation imposed by the hash function, $\tilde{q}_\cup$ is a uniform random sample of the distinct union $\vec{t}_1 \cup \ldots \cup \vec{t}_n$. An example is shown in Figure 2. Every prefix of a random permutation of the conceptual binary matrix that represents universe $\mathcal{D}$ and membership of set ids in inverted lists $\vec{t}_i$, is a random sample over $\mathcal{D}$.

Up to this point, we have constructed a uniform random sample, we can compute the score of all sampled sets with the query, and we know the size of the sample union. The final piece of the puzzle is to estimate the distinct number

Algorithm 2: HS (selectivity estimation algorithm).

Input : Query $q = \{t_1, \ldots, t_n\}$, Samples $\tilde{t}_1, \ldots, \tilde{t}_n$, Threshold τ
Output: Estimated number of sets with $\mathcal{I}(q, s) \geq \tau$
Let $A_S = 0$, $h_m = 0$
Compute $\tilde{q}_\cup = \tilde{t}_1 \cup \ldots \cup \tilde{t}_n$
forall $s \in \tilde{q}_\cup$ **do**
  If $\mathcal{I}(q, s) \geq \tau$, $A_S += 1$
  $h_m = \max(h_m, h(s))$
end
$D = |\mathcal{P}|(|\tilde{q}_\cup|_d - 1)/h_m$
Return $A_S \cdot \frac{D}{|\tilde{q}_\cup|_d}$

of set ids $|q_\cup|_d$, needed for scaling up the sample result. As already mentioned, maintaining separate synopses to this end will consume valuable space budget. The elegance of the consistent sampling algorithm described above is that the list samples themselves can be used for estimating the distinct number of ids in the union of an arbitrary combination of lists with high accuracy and probabilistic error bounds, by using the K-Minimum Values algorithm (KMV) proposed in [6]:

Theorem 1. *([6]) Given hash function $h : \mathcal{D} \to \mathcal{P}$, multiset S and letting h_r be the r-th smallest hash value in S, the quantity $D_r = |\mathcal{P}|(r-1)/h_r$ is an unbiased estimate of the distinct number of values in S. Specifically, given $0 < \delta < 1$, there exist ϵ dependent on r, s.t. $(1-\epsilon)|S|_d \leq D_r \leq (1+\epsilon)|S|_d$.*

Given that our samples contain all existing entries with hash values up to h_m in lists $\vec{t}_1, \ldots, \vec{t}_n$, we can immediately deduce the rank r of h_m in the sample union $\tilde{q}_\cup$, and hence directly estimate the distinct number of ids in $q_\cup$. The complete HS algorithm appears as Algorithm 2.

Notice that this algorithm needs to scan sample $\tilde{q}_\cup$ in order to estimate $|q_\cup|_d$, hence the speed up of this technique is directly proportional to the size of the sample used. For example, a 1% sample will result in approximately 100 times speed up with respect to any exact algorithm that examines a large portion of the inverted lists. Clearly, if we could design exact algorithms that examine only a small portion of the inverted index, selectivity estimation based on sampling would become obsolete. In practice, the fastest known algorithms for `TF/IDF` examine either 100% of the lists (fast sort merge joins, based on sorting by ids), or more than 70% of the lists on average (variants of NRA, based on sorting by partial weights). More details appear in [18].

Updating the samples is straightforward. For insertions, we hash the new entry and if the hash value is smaller equal to x we insert it in the sample. The Hashed Sampling algorithm, contrary to all other approaches that we have discussed so far, can also handle deletions gracefully. An entry deleted from a particular list, is simply deleted from the corresponding list sample, if it exists therein.

4.2 Theoretical Guarantees

Given that the resulting sample union of the random samples for an arbitrary query is always an $x\%$ sample of the union of the inverted lists of the query, we can give probabilistic error guarantees and space bounds using the VC-dimension of the problem [32]:

Theorem 2. *Let $\epsilon, \delta > 0$. Any random sample of size $O(\frac{1}{\epsilon^2} \ln \frac{1}{\epsilon^2 \delta})$ will provide an ϵ-approximate answer to the problem of selectivity estimation with probability of failure at most δ.*

Proof. The proof for the space bound of the HS technique is based on the work of Vapnik and Chervonenkis [32]. Some key definitions and concepts are presented next.

Definition 1. *([19]) A range space S is a pair (X, R) where X is a set and R is a set of subsets of X. Members of X are called points of S and members of R are called ranges of S.*

Definition 2. *([19]) Let $S = (X, R)$ be a range space and $A \subseteq X$ be a finite set of elements of S. Then $\Pi_R(A)$ denotes the set of all subsets of A that can be obtained by intersecting A with a range of S, i.e., $\Pi_R(A) = \{A \cap r : r \in R\}$. If $\Pi_R(A) = 2^{|A|}$, then we say that A is* shattered *by R. The* Vapnik-Chervonenkis dimension *of S is the smallest integer v such that no $A \subseteq X$ of cardinality $v+1$ is shattered by R.*

Definition 3. *([19]) Let $S = (X, R)$ be a range space and $A \subseteq X$ a finite subset of elements of S. For any $\epsilon \geq 0$ and $V \subseteq A$, V is an ϵ-approximation of A (for R) if for all $r \in R$, $|\frac{|A \cap r|}{|A|} - \frac{|V \cap r|}{|V|}| \leq \epsilon$.*

Theorem 3. *([32]) Let $S = (X, R)$ be a range space of VC-dimension v, $A \subseteq X$ be a finite set and $\epsilon, \delta > 0$. Then any random sample V of A formed by at least m independent draws from A is an ϵ-approximation of A for R with probability at least $1 - \delta$ for any $m \geq \frac{16}{\epsilon^2}(v \ln \frac{16v}{\epsilon^2} + \ln \frac{4}{\delta})$.*

It is very easy to see that a one dimensional range query has VC-dimension equal to 2 [7]. Since selectivity estimation of thresholded set similarity queries is equivalent to a one dimensional range counting query, the space bound follows. □

To find the total error of our selectivity estimation technique we also need to take into account the additive error from estimating the numerator of the scaling factor in Equation (5). Hence:

Theorem 4. *Let $\epsilon, \epsilon', \delta > 0$. Any random sample of size $O(\frac{1}{\epsilon^2} \ln \frac{1}{\epsilon^2 \delta})$, with probability of failure at most δ, in the worst case will provide answers within $(E - \epsilon|q_\cup|_d)(1-\epsilon') \leq \tilde{A} \leq (E + \epsilon|q_\cup|_d)(1+\epsilon')$, where E is the exact answer size, and $\tilde{A}$ the final selectivity estimate from HS.*

Proof. Theorem 2 suggests that $E - \epsilon|q_\cup|_d \leq A \leq E + \epsilon|q_\cup|_d$, where E is the exact answer and A the sample estimate. We are further estimating A within $(1-\epsilon')A \leq \tilde{A} \leq (1+\epsilon')A$, where ϵ' is the approximation error from Theorem 1. Combining the errors yields:

$$(E - \epsilon|q_\cup|_d)(1-\epsilon') \leq \tilde{A} \leq (E + \epsilon|q_\cup|_d)(1+\epsilon'). \quad (6)$$

□

It is important to notice here that estimation accuracy depends on two errors: that of estimating the frequency of query answers from the sample, and that of the K-Minimum distinct values estimation for scaling the results. Given any sample size $|S|$, from Theorem 1 and Theorem 2 we can

deduce that $\epsilon' \ll \epsilon$. Hence, as the size of the sample increases the error introduced by the distinct value estimation becomes exceedingly smaller than the error of frequency estimation.

Notice that the size of the sample for providing ϵ, δ guarantees for frequency estimation is not dependent on the size of the input. In practice, due to large hidden constants, sufficient sample sizes for accurate estimation will be much smaller than the ones suggested by the worst case bound given by the exact form of Theorem 2. Straightforwardly, as dataset sizes increase, the fact that the upper bound is constant is a very interesting result in sampling theory in itself. For example, a 5 million entries sample is the largest needed for providing $\epsilon = \delta = 0.01$ guarantees, irrespective of the input size. Hence, the theoretical bound with respect to the frequency estimation error makes sense in practice for dataset sizes of more than 100 million sets in this case, but our experiments show that even much smaller samples provide good accuracy in practice. On the other hand, the additive error due the estimation accuracy of the distinct value estimator always depends on the size of the sample used with respect to the size of the input.

It is interesting to note here that in theory and practice HS will work well irrespective of the underlying dataset distribution at hand, since it is a sampling based technique on what appears to always be a uniform distribution to the HS algorithm (the uniformity imposed by the hash function). We do not expect any bad or limit cases to appear in practice, if a good hash function is used (universal hash functions or a good heuristics), a fact attested by our empirical evaluation on three different datasets.

4.3 Improving accuracy

So far, we have not taken into account the semantic properties of the `TF/IDF` similarity for building the inverted list samples. By definition of idf, lists with small idfs have a very large size while lists with large idfs have small size. By taking a fixed $x\%$ sample of every list, we are allocating a much larger absolute space budget to low idf lists, since those lists are much larger. Low idf tokens result, on average, to small partial weights $w(t_i, s)$ and hence have low potency in the overall score. Hence, intuitively it appears that it would make sense to use stratification to divide our sample space into high and low idf tokens in order to capture a larger percentage of potent partial weights. The problem with stratification of inverted lists lies in computing a final estimate from the sample. Assume that we sample $z\%$ of high idf lists, and $y\%$ of low idf lists ($z > y$). First, we count the number of answers that exceed the threshold by using only the high idf lists. We scale up this result using the z sample. Then, we compute the number of answers that exceed the threshold by using at least one low idf list. We scale up this answer using the y sample. Finally, we add up the results. The drawback of this method is that it assumes uniformity of answers coming from solely high and solely low idf lists, which might not be true in general for all queries. Indeed, stratification did not yield any accuracy improvements in our empirical evaluation.

Another technique for improving sampling accuracy is post-stratification. Define a lattice structure that has, conceptually, one node for every combination of lists in query q, for a total of 2^n nodes (see Figure 3). Assign to every node in the lattice the set ids in $\tilde{q}_\cup$ that, first, exceed the query threshold and, second, have partial weights coming only from the lists corresponding to that node. Hence, nodes have mutually disjoint sets of ids, and the worst case maximum size of the instantiated lattice is at most $A_S \leq |\tilde{q}_\cup|$ nodes. Now consider the following procedure for scaling up the results from the sample. For every node, compute a node specific scale up factor which is calculated by taking into account only the lists that correspond to that node. Computing the scale up factor follows exactly the same reasoning as the one in HS. Compute as the final estimate the sum of the contributions from all nodes. The reasoning behind post-stratification is that as the number of instantiated lattice nodes increases, each node provides an independent estimate for a smaller subset of the space. Notice that a very small number of lattice nodes will be instantiated in practice. In order for a given set to exceed the threshold, it has to appear in at least a certain number of lists. Our experimental evaluation suggests that in practice very few queries result in a lattice with more than 4 nodes. To conclude, post-stratification does not hurt performance, and it can potentially provide some benefit for certain distributions of elements among lists, hence it should always be used. For our test data, post-stratification resulted in marginal benefits.

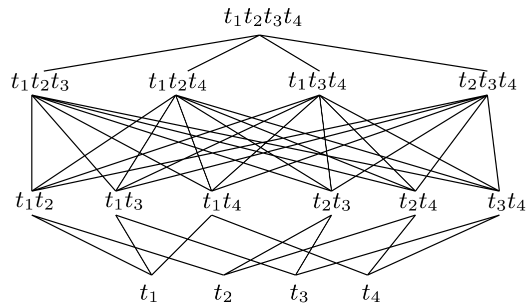

Figure 3: Post-stratification lattice for a query with four tokens. The worst case maximum instantiated lattice size will be $A_S \leq |\tilde{q}_\cup|$.

4.4 Improving performance

Given that low idf lists on average contain a large number of sets and at the same time they do not contribute significantly to the overall score, we can exclude such lists from processing, in order to improve performance. A principled way of choosing which lists to exclude appeared in [18], and we briefly repeat it here. Let query q with length $len(q)$, consisting of n tokens with idfs $\text{idf}(t_1) \geq \text{idf}(t_2) \geq \ldots \geq \text{idf}(t_n)$ (i.e., we sort tokens in decreasing idf order). Assume also that every list is sorted by decreasing partial weight, and let tf universally equal 1 for all list entries for simplicity. It is not hard to see that in order for a set s contained in list $\vec{t_i}$ to exceed threshold τ, provided that s does not appear in any list $\vec{t_j}, j < i$, it must have length ([18]):

$$len(s) \leq \lambda_i = \frac{\tau}{len(q)} \sum_i^n \text{idf}(t_i). \quad (7)$$

Thus, if the top element of $\vec{t_i}$ has weight $w < w_{\lambda_i}$, we can avoid processing the elements of this list. Our experimental evaluation shows that this optimization yields tremendous performance benefits in practice, without affecting estimation accuracy.

5. COMPARISON OF ALTERNATIVES

We have considered four main alternatives in the preceding analysis. Here we concisely compare those alternatives.

The first approach considered is simple uniform random sampling over the whole dataset. This approach gives theoretical guarantees on estimation accuracy (see Theorem 2), but the estimator cannot handle updates and limits the available space budget for storing samples in order to store the actual sets needed for score computation (the space needed per sample is not constant and depends on the average size of sets for a given application). In addition, given that the sample is constructed over the whole dataset and not over query specific only samples, it will yield estimates of very large variance.

The second approach discussed is to draw independent samples from each inverted list, and use only query specific list samples for selectivity estimation. Performance wise this method will be equivalent to exact evaluation since in order to scale up the sample estimate it needs to compute the distinct number of ids in the query lists, which requires a complete scan of the lists. Hence, this method is not competitive. Alternatively, we could use distinct value estimation synopsis in addition to the list samples in order to avoid scanning the inverted lists, but the proposed HS method subsumes this approach by building a combined sample/distinct value estimation synopsis per list.

The third technique discussed is to build query samples dynamically using reservoir sampling. This method does not provide any theoretical guarantees since the resulting sample is biased due to the presence of duplicates. It also requires storing distinct value estimation synopsis in addition to building the samples, and hence, is subsumed by HS.

HS provides concise theoretical guarantees on the size of the sample needed to achieve a given estimation error. It is the only approach that can support insertions and deletions. It builds specialized list samples that can be used both for producing a uniform random sample of the query lists on the fly, and for distinct value estimation. Resulting samples have small variance since they are built over query specific samples only. It utilizes the partial weight information contained in the sampled lists to compute scores, and hence does not need to store the actual sets in the sample. Thus, HS consumes smaller space per sample than simple random sampling (constant space in contrast to simple random sampling), and hence, for the same allotted budget $|S|$ it has better estimation accuracy (smaller ϵ).

6. EXPERIMENTS

First, we show that the straightforward solutions do not work at all in practice. Then, we contact a comprehensive empirical evaluation of the HS algorithm, on real datsets.

6.1 Setup

For our evaluation we use three real datasets; the DBLP citation database [24], the IMDB movie database [20], and the YellowPages listings [3]. More specifically, we use the Author/Paper association table from DBLP that contains approximately 2,500,000 pairs, the Actor/Movie association table from IMDB that contains approximately 7,000,000 pairs, and the Business Listing table from YellowPages that is significantly larger than IMDB (details are proprietary and cannot be disclosed). We decompose strings into sets of 3-grams and computed similarities using the popular `TF/IDF` score function. The average author and actor set size for DBLP and IMDB is 15. First, we build one inverted list per 3-gram on secondary storage. Then, we construct the corresponding list samples, according to a pre-defined space budget. We experiment with space budgets corresponding to 1%, 5%, and 10% of the total size of the inverted lists. We use the heuristic SHA1 hash function to create the samples in practice. All experiments are run on a four dual core Intel(R) Xeon(R) CPU 2.66 GHz, with 16 GB of main memory. We implement all algorithms in C++.

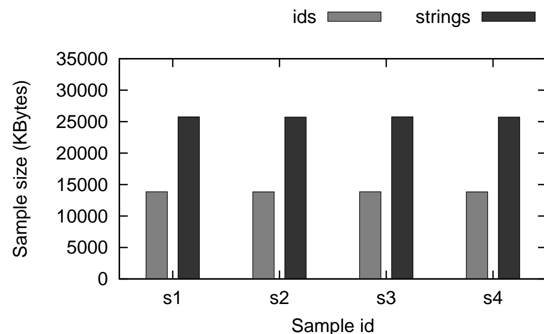

Figure 4: Space differential between id and string samples for DBLP dataset.

We draw queries from the actual data uniformly at random. Each query set contains 100 queries with a certain number of answers for a certain similarity threshold. More specifically, we choose queries that have between 100 to 200, 200 to 300, and 300 to 400 answers, for similarity thresholds equal to $0.4, 0.6$ and 0.8; this results in a total of 9 query sets. We avoid queries with a smaller number of answers, since for numbers of small magnitude average relative errors are not meaningful from a practical point of view (e.g., an estimated answer of 2 versus an exact answer of 1, yields 50% error).

We evaluate the proposed algorithms in terms of estimation accuracy (average relative errors with respect to the exact answers), and wall-clock time (CPU and I/O cost). We perform post-stratification for all algorithms. As a baseline comparison we use the optimized TA/NRA exact evaluation algorithms for the `TF/IDF` measure, proposed in [18]. We expect the speed up of HS to be proportional to the sample size, irrespective of the algorithm used, since the selectivity estimators simply need to scan the whole sample (usually small enough to be prefetched or buffered in main memory), while the exact algorithms need to read a large portion of the disk resident inverted lists. We test all algorithms for varying thresholds, query answer sizes, and available budget. In all of our experiments we average results over 100 queries and 4 independent runs per algorithm.

6.2 Straightforward solutions

In what follows we show results for the DBLP dataset. Results for IMDB and YellowPages followed the same trends overall. Furthermore, the default experimental parameters are 5% budget (with respect to the total size of the inverted index), 100 to 200 answers per query for a 0.4 threshold.

First, we show that storing the actual strings, instead of 8 byte string id/partial weight pairs in the samples reduces

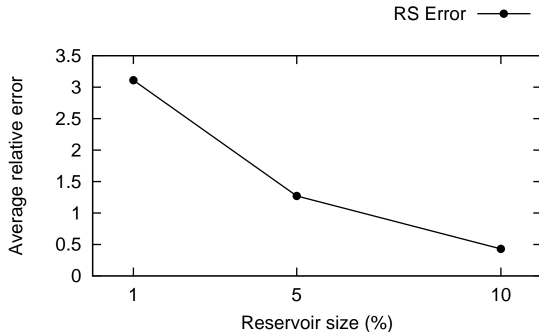

Figure 5: Accuracy of the dynamic reservoir sampling algorithm.

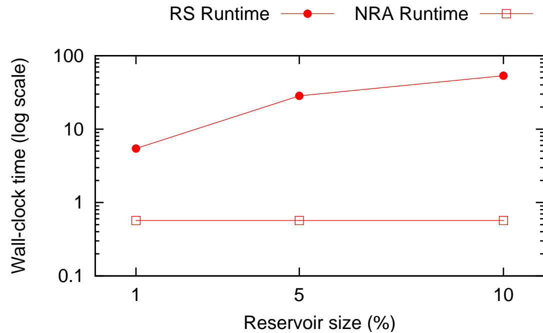

Figure 6: Runtime cost of the dynamic reservoir sampling algorithm.

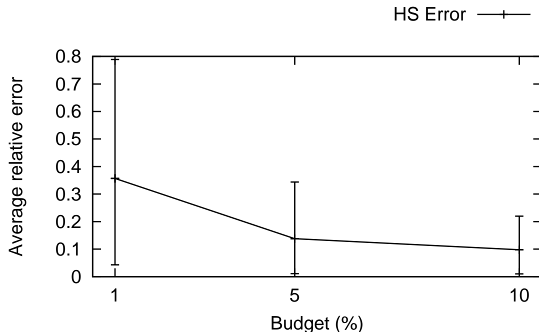

Figure 7: Accuracy of the HS algorithm as a function of available budget.

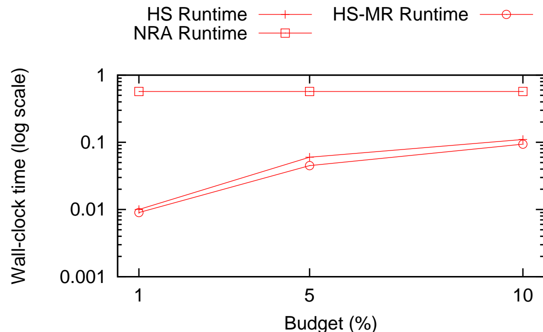

Figure 8: Runtime cost of the HS algorithm as a function of available budget.

the effective size of the sample by almost 50%. We created 4 independent 5% samples of string ids, computed the size of the resulting samples, then replaced each id with the corresponding string and measured the increase in size. Figure 4 shows that the size of the string sample was consistently 1.85 times larger than the size of the ids sample. This directly implies that simple random sampling cannot compare to HS, since for the same available space budget HS will be able to store almost twice as many samples than random sampling. For completeness, we evaluated the performance of the naive sampling approach. This technique yields consistently larger than 40% errors in all cases, since it does not build query specific samples.

We evaluate the cost and accuracy of the Reservoir Sampling technique discussed in Section 3.2. Recall that reservoir sampling will inadvertently create a biased sample. Hence, we do not expect estimates of high accuracy. At the same time, since any dynamic sampling technique has to retrieve the actual strings from storage, runtime performance will suffer since a large number of random I/Os will need to be performed. Figures 5 and 6 show the accuracy and the runtime cost of reservoir sampling with reservoir sizes equal to $1\%, 5\%$ and 10% of the total size of the relevant inverted lists for a given query. Clearly, this technique yields inaccurate estimates, and has an extremely high processing cost. As a baseline comparison, we plot the runtime cost of producing exact answers using the fast NRA algorithm [18] on the same plot (notice that the x-axis is not relevant for this algorithm). The runtime plot is in log scale.

6.3 Hashed Sampling

In the rest, we evaluate the HS algorithm only, since it is the only viable solution. Figures 7 and 8 plot accuracy and cost as a function of space budget. In the same graphs we superimpose the cost of producing exact answers using the fast NRA algorithm. As expected, the runtime cost of HS increases as the sample size increases, but is up to two orders of magnitude faster than NRA. We also run the TA algorithm for completeness; it's runtime was six times slower than NRA on average, due to the increased number of random accesses performed. HS achieves very high quality answers for small sample sizes. A 5% sample of the inverted lists of the DBLP dataset consumes a total of 20MB. One can maintain this sample in main memory for faster processing. The runtime of main memory resident HS is plotted as HS-MR in the same figure. Clearly, the disk and the operating system do a great job in pre-fetching and buffering the sample, hence we observe a very small benefit in practice. In this figure we also show the 5th and 95th percentiles of relative errors (as error bars in the graph), as an indication of the variance of the proposed algorithm. As the budget size increases, the variance decreases proportionately.

Figures 9 and 10 plot accuracy and runtime performance of the same algorithms, as a function of varying query thresholds. The accuracy of HS deteriorates somewhat for increased thresholds, but remains consistently below 17%. We also see an interesting trend here with regard to runtime performance. The cost of running the algorithms drops sharply as thresholds increase, which is expected since larger thresh-

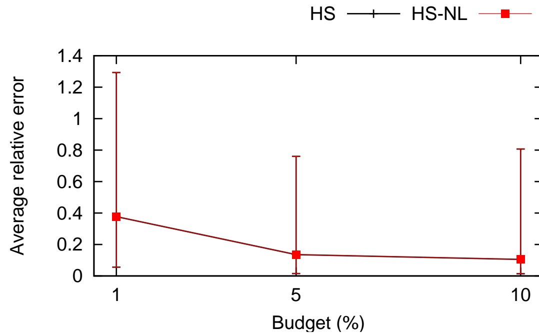

Figure 13: Comparison between HS with and without lattice based post-stratification.

olds imply fewer potential candidates exceeding the threshold, and faster termination of the algorithms. HS is up to two orders of magnitude faster than NRA.

Figures 11 and 12 plot accuracy and runtime performance of HS as a function of query answer size. We can see that the error drops sharply as the answer size increases. Straightforwardly, any selectivity estimation algorithm should be able to more accurately estimate queries of low selectivity. Moreover, the runtime cost of HS remains constant, which is expected since it is dominated by the cost of computing $|q_{\cup}|_d$ (which requires reading the whole sample). The cost of NRA slightly increases with increasing number of answers since first, a larger number of list elements need to be accessed as the query becomes less selective and second, maintaining the candidate set becomes more expensive.

6.4 Improving accuracy

Next, we evaluate the benefits of using the lattice based post-stratification algorithm, with respect to estimation accuracy. We profiled all strings in the DBLP dataset with respect to the number of nodes they instantiate during post-stratification. Overwhelmingly, in practice queries instantiate less than 5 lattice nodes. Out of 700,000 queries, only 132 queries with answer size between 100 and 200 strings for threshold 0.4, had more than 4 lattice nodes. We constructed a special query set out of those 132 queries in order to exemplify the potential benefits of post-stratification. Results with (HS) and without (HS-NL) post stratification as a function of budget space, are shown in Figure 13. Post-stratification had only a marginally positive effect on accuracy. We observed exactly the same trends for the IMDB and YellowPages data.

The next experiment evaluates the benefits of using high and low idf list stratification. First, we need to decide an idf cutoff threshold between high and low idf lists. Figure 14 plots the cumulative size of lists up to a certain idf. We can see that lists with idf smaller than 8 add up to almost 50% of the total inverted lists size. By sampling those lists at a 3% rate, we are able to sample high idf lists at a 9% rate, for an overall 5% sample. Figure 15 shows the results of stratification. We can see that in our setting, using the semantics of idf to perform stratification is counter-productive, yielding 80% errors. We profiled a large number of queries for the DBLP dataset that yield between 100 and 200 answers using a 0.4 threshold, and deduced that on average 64% of the tokens of a given query had idf smaller than 8. Thus, most queries have to use the relatively small low idf sample. Exactly the same trends are observed in the IMDB and YellowPages corpora.

6.5 Improving performance

Next we evaluate the performance benefits of excluding low idf lists from processing, as well as impact on accuracy. The results are shown in Figures 16 and 17, where HS-LI denotes the version of the algorithm that processes the low idf lists normally. We superimpose on the graph the performance of the NRA-LI algorithm (i.e., without the low idf optimization). We can see that the low idf optimization yields significant performance benefits, without hurting accuracy. This graph also exemplifies our claim that the performance of the selectivity estimation techniques is mainly dependent on the size of the sample. Notice that as the size of the sample increases, the runtime cost of the HS-LI algorithm increases (due to the large size of the low idf sampled lists that have to be processed).

The last experiment (Figures 18 and 19) tests the distinct value estimation accuracy of the KMV synopsis. For these results we run the HS algorithm once more, but this time we compute the exact number of distinct set ids contained in the inverted lists of each query, before scaling up the answers (HS-NE). We can see that the distinct value estimation does not hurt accuracy, yielding almost identical answers. This verifies our intuition that the error rate of KMV is dominated by the error rate of frequency estimation using the sample, as the size of the sample increases. We also show the overwhelming cost of having to scan the inverted lists in order to compute the exact number of distinct ids in the inverted lists of the query (in log scale).

7. RELATED WORK

The problem of selectivity estimation has been studied extensively in the context of query optimization. The main approaches utilize either random sampling or histograms. Histograms are discussed in [27]. Histograms work well for numerical attributes. One could use value-range histograms in our setting, by sorting strings lexicographically. However, this would not produce good estimates, since lexicographical proximity could be small even if `TF/IDF` (or edit distance) similarity is large. A survey of existing work on random sampling appears in [26]. Haas and König [17] and Chaudhuri et al [11] were the first to address sampling efficiency with respect to random accesses for retrieving the actual data. Our approach is inspired by the same problem, but in our solution we make efficient use of available index structures to build specialized samples that eliminate the need to access the data.

Selectivity of approximate string matching queries has been addressed in the past. Jin and Li [22] propose selectivity estimation techniques for the edit distance function based on clustering and histograms. Their approach can be extended, with some limitations, to other distance functions as well, but dynamically maintaining the estimator is expensive and results in deterioration of estimation accuracy over time, especially if the cluster of strings begins to shift over time. Sahinalp et al. [28] use a two step approach that also uses clustering of strings within a certain edit distance. In the first step, clusters not relevant to the query are pruned. In the second step the remaining can-

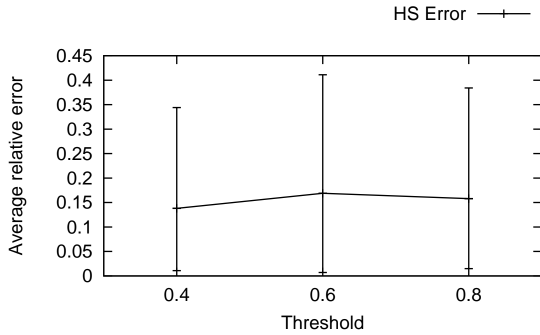

Figure 9: Accuracy of the HS algorithm as a function of query thresholds.

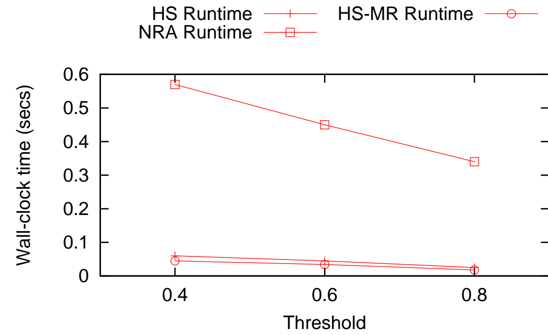

Figure 10: Runtime cost of the HS algorithm as a function of query thresholds.

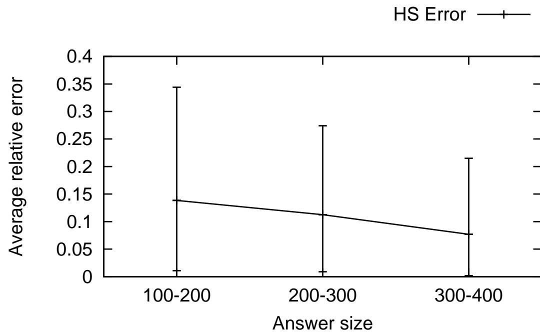

Figure 11: Accuracy of the HS algorithm as a function of query answer size.

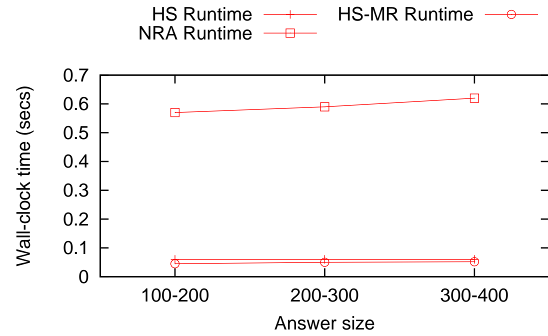

Figure 12: Runtime cost of the HS algorithm as a function of query answer size.

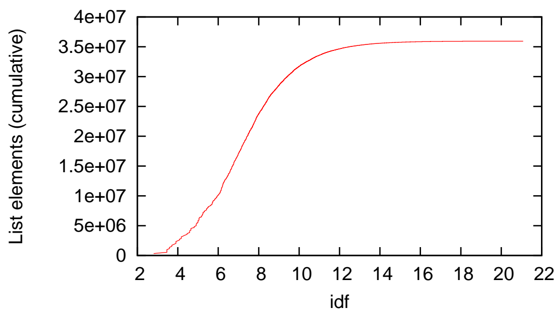

Figure 14: Cumulative number of entries contained in lists of tokens below a certain idf.

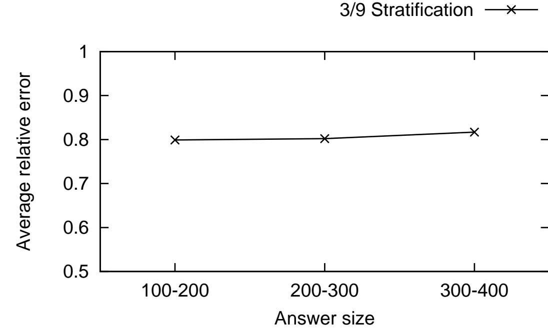

Figure 15: Effects of using stratification across high and low idf lists.

didate strings are scanned and the exact edit distances are computed. The pruning technique works well only if the database contains a large number of small string neighborhoods. An improved estimator, specifically for edit distance, appears in [25], where the authors use q-gram signatures as a compact representation of the dataset. A different approach, based on wildcards, is presented in [23] for estimating selectivity of approximate string matching with low edit distance.

Guha et al [16] propose a sampling based technique for selectivity estimation of join queries under `TF/IDF` cosine similarity. This technique constructs samples of the conceptual binary matrix corresponding to the inverted lists of a given relation, similarly to our setting (see Figure 2). Then, it estimates the selectivity of the join by using the product of the sample matrices corresponding to the joined relations. In this work, efficiency was not a concern; rather the sampling methodology was discussed. Our work concentrates on ways of constructing consistent samples that enable very efficient estimation using the information contained in the inverted lists. Our approach is faced with a different set of problems as well. For example, leveraging the partial weight information in the inverted lists, and performing distinct value estimation to scale up the results. Our estimators can be used for joins in vein similar to [16]. In general, using sampling to estimate join results does not always resulting in accurate estimates. Nevertheless, it is an interesting open problem whether our hashing based samples

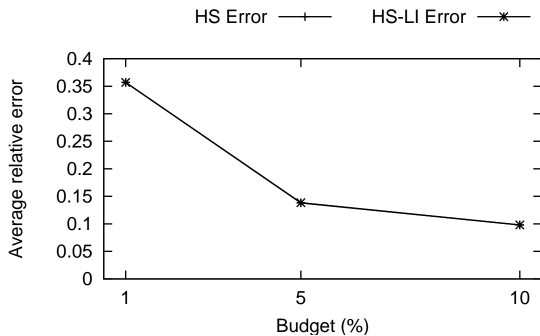

Figure 16: Effect of low idf lists on accuracy.

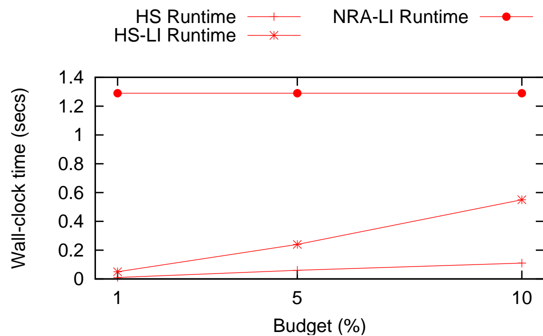

Figure 17: Effect of low idf lists on runtime cost.

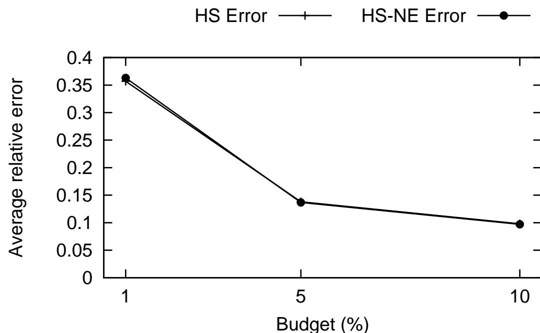

Figure 18: Distinct value estimation impact on overall accuracy.

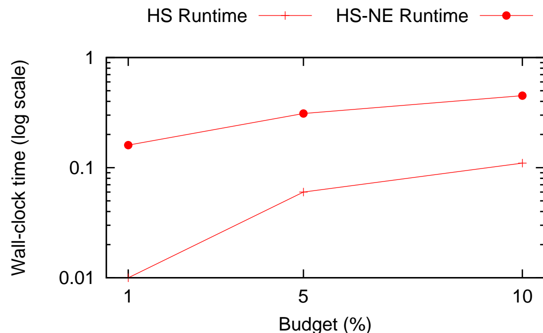

Figure 19: Distinct value estimation impact on runtime cost.

provide some advantages over the straightforward sampling techniques presented in [16].

Another approach for selectivity estimation of TF/IDF based approximate string matching appears in [31]. The authors propose estimating various statistical properties of the dataset that can be used in order to eventually estimate selectivity. This approach cannot support updates, does not provide any guarantees, and in practice, it achieves close to 40% errors and hence cannot compete with HS.

Another related problem is that of substring selectivity, or estimating the number of database strings that have a given query string as a substring. Quite a few techniques have been proposed for this problem [21, 12], but they cannot be used for providing estimates for selectivity on approximate string matching queries on any distance measure.

8. CONCLUSION

We explore sampling techniques for selectivity estimation of set similarity queries using traditional weighted similarity measures, like TF/IDF. We show that straightforward approaches will not work better then exact solutions in practice, or will result in biased samples of very low accuracy. We propose a novel algorithm based on consistent sampling through hashing, that results in answers of very high quality. We also show experimentally that our approach works very well for a large range of real datasets. As future work, we plan to investigate if our techniques can be applied to other similarity measures, like HMM and Jaccard.

9. REFERENCES

[1] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM (JACM)*, 46(5):667–683, 1999.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. of Very Large Data Bases (VLDB)*, pages 918–929, 2006.

[3] AT&T Inc. Business listings from YellowPages.com. proprietary.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, May 1999.

[5] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[6] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. of ACM Management of Data (SIGMOD)*, pages 199–210, 2007.

[7] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the vapnik-chervonenkis dimension. *Journal of the ACM (JACM)*, 36(4):929–965, 1989.

[8] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[9] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In *Proc. of ACM*

Management of Data (SIGMOD), pages 353–364, 2007.

[10] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 268–279, 2000.

[11] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proc. of ACM Management of Data (SIGMOD)*, pages 287–298, 2004.

[12] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 227–238, 2004.

[13] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. of International Conference on Data Engineering (ICDE)*, page 5, 2006.

[14] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[15] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proc. of Very Large Data Bases (VLDB)*, pages 491–500, 2001.

[16] S. Guha, N. Koudas, D. Srivastava, and X. Yu. Reasoning about approximate match query results. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 8–18, 2006.

[17] P. J. Haas and C. König. A bi-level bernoulli scheme for database sampling. In *Proc. of ACM Management of Data (SIGMOD)*, pages 275–286, 2004.

[18] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *Proc. of International Conference on Data Engineering (ICDE)*, 2008.

[19] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. In *Proc. of Annual Symposium on Computational Geometry (SCG)*, pages 61–71, 1986.

[20] IMDB. IMDB database. http:// www.imdb.com/interfaces.

[21] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal*, 9(3):214–230, 2000.

[22] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *Proc. of Very Large Data Bases (VLDB)*, pages 397–408, 2005.

[23] H. Lee, R. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proc. of Very Large Data Bases (VLDB)*, pages 195–205, 2007.

[24] M. Ley. DBLP database. http://dblp.uni-trier.de/xml.

[25] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Transactions on Database Systems (TODS)*, 32(2):12, 2007.

[26] F. Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, CA, 1993.

[27] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of ACM Management of Data (SIGMOD)*, pages 294–305, 1996.

[28] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 125–136, 2003.

[29] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM Knowledge Discovery and Data Mining (SIGKDD)*, pages 269–278, 2002.

[30] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. of ACM Management of Data (SIGMOD)*, pages 743–754, 2004.

[31] S. Tata and J. M. Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *SIGMOD Record*, 36(2):7–12, 2007.

[32] V. N. Vapnik and A. Y. Chrervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *SIAM: Theory of Probability and its Applications (TPA)*, 16(2):264–280, 1971.

[33] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.