Incremental Maintenance of Length Normalized Indexes for Approximate String Matching

Marios Hadjieleftheriou
AT&T Labs – Research
Florham Park, NJ, USA
marioh@research.att.com

Nick Koudas
University of Toronto
Toronto, ON, Canada
koudas@cs.toronto.edu

Divesh Srivastava
AT&T Labs – Research
Florham Park, NJ, USA
divesh@research.att.com

ABSTRACT

Approximate string matching is a problem that has received a lot of attention recently. Existing work on information retrieval has concentrated on a variety of similarity measures (`TF/IDF`, `BM25`, `HMM`, etc.) specifically tailored for document retrieval purposes. As new applications that depend on retrieving short strings are becoming popular (e.g., local search engines like YellowPages.com, Yahoo!Local, and Google Maps) new indexing methods are needed, tailored for short strings. For that purpose, a number of indexing techniques and related algorithms have been proposed based on length normalized similarity measures. A common denominator of indexes for length normalized measures is that maintaining the underlying structures in the presence of incremental updates is inefficient, mainly due to data dependent, precomputed weights associated with each distinct token and string. Incorporating updates usually is accomplished by rebuilding the indexes at regular time intervals. In this paper we present a framework that advocates lazy update propagation with the following key feature: Efficient, incremental updates that immediately reflect the new data in the indexes in a way that gives strict guarantees on the quality of subsequent query answers. More specifically, our techniques guarantee against false negatives and limit the number of false positives produced. We implement a fully working prototype and illustrate that the proposed ideas work really well in practice for real datasets.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

General Terms

Algorithms

1. INTRODUCTION

A variety of applications deal with short strings. Examples include directory search that retrieves business listings relevant to a short query string (e.g., YellowPages.com, Yahoo!Local, Google Maps) and data cleaning and record linkage applications on relational data that try to match records across tables and databases (e.g., employee names and addresses) [16, 8]. Given that queries often contain spelling mistakes and other errors, and stored data have inconsistencies as well, effectively dealing with short strings requires the use of specialized approximate string matching indexes and algorithms. Previous work on information retrieval has focused mostly on document retrieval (e.g., Google [11], Lucene [1], FAST [10]). Although fundamentally documents are long strings, these approaches make assumptions that in general are not true when dealing with shorter strings. For example, the frequency of a term in a document might suggest that the document is related to a particular query or topic with high probability, while the frequency of a given token or word in a string does not imply that a longer string (containing more tokens) is more similar to the query than a shorter string. Or the fact that shorter documents are preferred over longer documents (for parsimony) conflicts with the fact that in practice for short queries the vast majority of the time users expect almost exact answers (answers of length similar to the length of the query). This is compounded by the fact that for short strings length does not vary as much as for documents in the first place, making some length normalization strategies ineffective. Moreover, certain other properties of short strings enable us to design very fast specialized approximate string matching indexes in practice [2, 22, 13, 18].

In many applications it is not uncommon to have to execute multiple types of searches in parallel in order to retrieve the best candidate results to a particular query, and use a final ranking step to combine the results (e.g., almost exact search versus sub-string search, ignore special characters search, full string search or per word search, 2-grams or 3-grams or 4-grams, edit distance versus `TF/IDF` search). From our experience, when dealing with short string queries, the majority of users are looking for almost exact matches (e.g., the queries 'wallmart', 'wal-mart', and 'wal mart' should all return the correct listing 'walmart', instead of 'walgreens wallpaper mart'). In that respect, we would like to build indexes that can retrieve almost exact matches as fast as possible, and revert to more"fuzzy" (and hence slower) searches as a subsequent step, in order to be able to provide interactive responses for the majority of queries (e.g., in the form of a text completion box as the user is typing the query).

Recently, the work in [13] showed that using L_2 length normalization when building the inverted indexes enables us to retrieve almost exact matches almost for free by using very aggressive pruning strategies on the inverted lists. Nev-

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

ertheless, the drawback of this approach is that the indexes are expensive to construct and they do not support incremental updates. Generally speaking, even though various types of length normalization strategies have been proposed in the past, approaches that have strict properties that can enable aggressive index pruning are hard to maintain incrementally, while simpler normalization methods are easier to maintain but suffer in terms of query efficiency and result quality, yielding slower answers and significantly larger (i.e., fuzzier) candidate sets.

A key issue that we have to deal with in a real system is that data is continuously updated. A small number of updates to the dataset would necessitate near complete recomputation of an L_2 normalized index, since L_2 is sensitive to the total number of records in the dataset, and the distribution of tokens (n-grams, words, etc.) within the strings. Given that datasets tend to contain tens of millions of strings and that strings could be updated on an hourly basis, recomputation of the indexes can be prohibitively expensive. In most practical cases, updates are buffered and the indexes are rebuilt on a weekly basis. Index recomputation typically takes up to a few hours to complete. However, the online nature of some applications necessitates reflecting updates to the data as soon as possible. Hence, being able to support incremental updates as well as very efficient query evaluation are critical requirements.

In [15] two techniques were proposed for enabling propagation of updates to the inverted indexes. The first was *blocking* the updates and processing them in batch. The second was *thresholding* updates and performing propagation in multiple stages down the index, depending on the update cost one is willing to tolerate. That work presented heuristics that perform well in practice, based on various observations about the distribution of tokens in real data, but it did not provide any theoretical guarantees with respect to answer accuracy while updates have not been propagated fully. In our work we develop an update propagation framework on length normalized inverted indexes that incorporates updates very efficiently and at the same time provides strict guarantees on the precision of query answers on the updated index. We show that our algorithms can propagate a batch of daily updates in a matter of minutes, rather than the few hours that it would take to rebuild the index, while the updated index can be used to answer queries very efficiently with no false negatives and a small number of false positives with respect to the answers that would be provided by fully propagated updates. We study the properties of the proposed updating framework theoretically using a rigorous analysis, and illustrate its efficiency on real datasets and updates, through a comprehensive experimental study.

Section 2 discusses length normalized inverted indexes in detail. Section 3 presents a high level description of the proposed update propagation framework. In Section 4 we conduct a detailed theoretical analysis of the approach. Section 5 discusses the update propagation algorithm in detail. Section 6 presents a thorough experimental evaluation. Related work is discussed in Section 7. Finally, Section 8 concludes the paper.

2. PRELIMINARIES

Indexes for approximate string matching are mostly based on token decomposition of strings (e.g., into n-grams or words) and building inverted lists over these tokens. Then, similarity of strings is measured in terms of similarity of the respective token sets (e.g., by using the vector space model to compute cosine similarity, or positional tokens to estimate edit distance). Consider strings "Walmart" and "Wal-mart". We can decompose the two strings in 3-gram sets {'Wal'/1, 'alm'/2, 'lma'/3, 'mar'/4, 'art'/5} and {'Wal'/1, 'al-'/2, 'l-m'/3, '-ma'/4, 'mar'/5, 'art'/6}. The two sets have three 3-grams in common, one of which matches in position exactly, while the other two match the position within distance one. Using the two sets we can compute both `TF/IDF` based cosine similarity scores and an upper bound on the edit distance between the two strings.

Formally, consider a collection of strings D, where every string consists of a number of tokens from universe $\mathcal{U}$. For example, let string $s = \{t^1, \ldots, t^n\}, t^i \in \mathcal{U}$. Let $df(t^i)$ be the total number of strings in D containing token t^i and N be the total number of strings. Then:

$$idf(t^i) = \log_2 (1 + N/df(t^i)). \quad (1)$$

Another popular definition of idf is based on the Okapi `BM25` [20] formula:

$$idf(t^i) = \log_2 \frac{N - df(t^i) + 0.5}{df(t^i) + 0.5}. \quad (2)$$

The L_2 length of string s is computed as

$$L_2(s) = \sqrt{\sum_{t^i \in s} idf(t^i)^2}, \quad (3)$$

and one can also compute simpler lengths based on $L_1(s) = \sum_{t^i \in s} idf(t^i)$, or the number-of-tokens normalization $L_0(s) = \sum_{t^i \in s} 1$. Define the L_2 normalized `IDF`, `BM25` similarity of strings s_1 and s_2 as:

$$\mathcal{S}_2(s_1, s_2) = \sum_{t^i \in s_1 \cap s_2} \frac{idf(t^i)^2}{L_2(s_1) L_2(s_2)}, \quad (4)$$

assuming that for short strings the token frequency of the majority of tokens is equal to 1. L_2 normalization forces similarity scores in the range $[0, 1]$. We can define score functions with similar properties for L_1. On the other hand, with L_0 normalization the range of similarity scores is bounded only by the maximum string length in the dataset. Furthermore, for L_2 an exact match to the query always has similarity equal to one (it is the best match). On the other hand, for L_0 an exact match to the query is not always the best match. In other words, exact matches can have similarity with the query smaller than non-exact matches. This is easy to see with an example. Referring to Figure 1, assume that $q = \{t^1, t^2, t^3\}$ with $L_0(q) = 3$. Let similarity $\mathcal{S}_0(q, s) = \sum_{t^i \in q \cap s} idf(t^i)^2 / L_0(q) L_0(s)$. Let data strings $s_1 = \{t^1\}$ and $s_4 = \{t^1, t^2, t^3\}$. Then, $\mathcal{S}_0(q, s_1) = 100/3 > \mathcal{S}_0(q, s_4) = 168/9$. It is easy to see that irrespective of the actual similarity function used we can always construct such examples. Given that we are interested in identifying almost exact matches fast, the benefits of using L_2 normalization are evident.

Consider an approximate string matching query that, given a query string q, retrieves from the dataset all strings $s \in D : \mathcal{S}_\star(q, s) \geq \tau$. It can be shown that by using $\mathcal{S}_2$ similarity:

THEOREM 1 (LENGTH BOUNDEDNESS [13]). *Given query string q, string s and threshold τ, if $\mathcal{S}_2(q, s) \geq \tau$ it follows that $\tau L_2(q) \leq L_2(s) \leq \frac{L_2(q)}{\tau}$.*

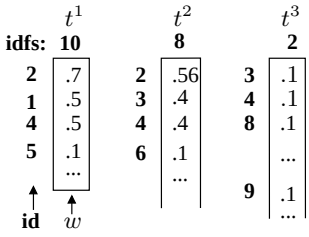

Figure 1: Inverted lists sorted by decreasing token contribution in the overall score.

Clearly, by using Theorem 1 we can immediately prune all strings whose lengths fall outside the given bounds, hence enabling very aggressive pruning of indexed strings. Notice that given the length of the query string, the theorem determines which database strings have lengths either too short or too long to actually exceed the user defined similarity threshold. This is irrespective of the number of tokens these strings have in common with the query, or the magnitude of the idfs of the common tokens. On the other hand, it can be shown easily using counter-examples, that no such property holds in the vector space model for L_0. Hence, the benefits of using L_2 in terms of query answering efficiency are also evident (experimental evidence of which appears in Section 6).

Typical approximate string matching indexes consist of inverted lists built on the tokens in $\mathcal{U}$. Formally, let:

$$w(s, t^i) = idf(t^i)/L_\star(s), \tag{5}$$

be the partial score contribution of token t^i in $\mathcal{S}_\star(q, s)$, independent of q. We construct one inverted list per token $t^i \in \mathcal{U}$, that consists of one pair $\langle s, w(s, t^i)\rangle$ per string s containing t^i. An example of inverted lists corresponding to three tokens t^1, t^2, t^3 appears in Figure 1, where data strings from the database are associated with a unique identifier in the index, to save space.

Let $q = \{t^1, \ldots, t^n\}$ with length $L_\star(q)$. We can find all strings that exceed similarity score τ by scanning the inverted lists corresponding to $t^1, \ldots, t^n$ and computing $\mathcal{S}_\star(q, s)$ for all s. If lists are sorted by string id we can use multi-way merge sort based algorithms to compute string scores very fast. Still, these algorithms need to read lists exhaustively in order to find all strings with similarity larger than τ. Alternatively, assume that lists are sorted in decreasing $w(s, t^i)$ order (and secondarily in increasing string id order). Given monotonic similarity functions (e.g., Equation (4)), we can use TA/NRA [9, 13] algorithms to compute the scores incrementally, and terminate before exhaustively reading the lists. Moreover, when using L_2 lengths to build the lists, we can make use of Theorem 1 to restrict the part of the lists we scan within the window $\tau L_2(q) \leq L_2(s) \leq \frac{L_2(q)}{\tau}$ (since lists are implicitly sorted by increasing string lengths when they are sorted by decreasing partial weights w), dramatically limiting the total size of the lists we have to examine, hence improving efficiency of query answering.

An example is shown in Figure 1. Assume that query q consists of only three tokens t^1, t^2, t^3. In order to compute the similarity score of all database strings that have at least one token in common with the query we simply have to scan the three inverted lists corresponding to the query tokens and sum up the partial weights (multiplied by the query token weights as in Equation (4)). Notice that only string id 4 is contained in all three lists. There are two cases here. Either string 4 is the same as the query, or string 4 is a super-set of the query. By using the length of string 4 we can conclusively determine either case. Assuming that the partial weights in the lists have been computed using L_2 lengths, we can easily see that the length of string 4 is equal to $w(4, t^1) = 10/L_2(4) \Rightarrow L_2(4) = 10/0.5 = 20$, while the length of the query string is $\sqrt{10^2 + 8^2 + 2^2} = 12.96$. Now, given a similarity threshold τ and by using Theorem 1 we can compute tight partial weight ranges within each list that we have to examine, in order to deterministically find all strings with similarity larger than τ.

The extra cost of building the length normalized inverted lists stems from the need to sort the lists in decreasing order of partial weights. The prohibitive cost of supporting incremental updates stems from the fact that L_2 lengths depend on token idfs, idfs change as updates occur, and hence the length of any string may change even if the string itself did not change. In contrast, notice that with L_0 normalization the length of a string does not depend on the idfs of the string tokens, hence maintaining the lists is easier and less expensive. On the other hand, L_0 normalization is more relevant to document retrieval, where there is no focus on finding almost exact matches to query strings, L_0 does not offer the query answering efficiency advantages of L_2, and finally, the proposed lazy update propagation framework for L_2 will result in incremental updates that are as fast as fully propagated updates for L_0, and with an insignificant drop in query answer precision.

3. PROPAGATING UPDATES

3.1 Building from scratch

The easiest way to propagate a set of updates is to rebuild the inverted lists from scratch. For large datasets this is infeasible on a frequent basis since this is a very expensive operation, especially for length normalized indexes. In this section we give a detailed analysis of the construction process to clearly illustrate the costs associated with building these indexes.

The first step of the construction process is a linear scan over the data strings, pre-processing of the data to format it appropriately (make it case insensitive, remove special characters, stemming, etc.), extraction of tokens (n-grams or words) and computation of token frequencies. At the end of the first pass we can compute the idf of each token. After idfs have been computed, a second pass over the data is performed (after performing exactly the same pre-processing on the strings that occurred during the first pass).[1] During the second pass, the tokens of every string are generated once again, the normalized length of the string is computed (using the idfs from the previous step for L_2 normalization), and the sorted inverted lists are created. Notice that for L_0 normalization we only need to perform one pass over the data since the idfs are not needed for computing the length of the strings.

The goal of the second pass is to prepare the tokens for building sorted inverted lists. It should be mentioned here that if we are not willing to tolerate the cost of sorting the lists, we can always use unsorted lists and exhaustive scan

[1]Alternatively we can store the pre-processed strings in an intermediate file in case pre-processing is expensive.

Algorithm 3.1: CONSTRUCT(D)

for each $s \in D$:
  do { Pre-process s
  for each $t \in s : df(t) += 1$
for each $s \in D$:
  do { Compute $L_2(s)$
  for each $t \in s : append\langle s, w(s,t)\rangle tolist(t)$
for each t
  do { Sort $list(t)$ on w
  Build B-tree

Figure 2: Selection sort construction algorithm.

querying strategies instead of TA/NRA algorithms (with either L_0 or L_2), but for a steep penalty on query performance (as will be seen in Section 6). In the rest we concentrate only on sorted inverted lists. There are two ways to accomplish this task. The first is to prepare the tokens for external sorting (lexicographically by token and numerically by partial weight). After performing the external sort the final sorted file can be split easily into one inverted list per token (e.g., by bulk loading one B-tree [7] per token). The second option is to use a version of selection sort directly to secondary storage as tokens are being generated. The idea is to create one empty inverted list per token (with the potential of having to manage thousands of files at a time and thrashing the disk) and store tokens in their respective files as they get generated. Then, after all inverted lists have been populated, we scan and sort each inverted list by partial weight independently (either in main memory if the list is small or using external sorting if the list is large). Alternatively, we can directly create one B-tree per token and direct each token to the appropriate B-tree as it is generated, again with the adverse effect of thrashing the B-trees. A sample list construction algorithm is shown in Algorithm 3.1.

At first it might appear that using external sorting is more efficient than having to manage a very large number of open files or B-trees for the selection sort step. Nevertheless, there are several trade-offs that need to be taken into account. The cost of using buffered files is that files need to be flushed, closed and reopened on demand as main memory becomes full. The cost of directly inserting into B-trees is the random I/O incurred per B-tree insertion. The external sort approach has to read and write each token to disk multiple times (at least once in the beginning to create the initial buckets, a second time for the first merging step, and a third time for splitting the final sorted file and creating the B-trees), which can become prohibitive when dealing with tens of millions of short strings that get decomposed into several hundred million tokens. On the other hand, external sort will scale irrespective of the total available main memory size and dataset size. Our experiments show that when there are only a few lists (hundreds to a few thousand), selection sort directly to buffered files and subsequent bulk loading of B-trees outperforms external sorting on small data sets (up to tens of millions of tokens). However, when the number of lists is huge, or the data sets are very large (several hundred million tokens), external sorting is the fastest.

For example, indexing the 2.5 million author names in the DBLP dataset [17] produces 40 million 3-grams and 42 million 4-grams. For this dataset we have to manage approximately 24,000 inverted lists when using 3-grams and 161,000 for 4-grams. Clearly, as the number of lists explodes, it becomes more and more expensive to manage the buffered files needed for selection sort, while external sort uses a fixed amount of memory irrespective of the number of lists. On the other hand, for the Business Listings dataset (see Section 6) that generates approximately 365 million n-grams, selection sort is almost as efficient as external sorting.

ID	Author Name	# of papers
1	Michael Carrey	36
2	David DeWitt	33
3	Surajit Chaudhuri	31
4	Jeffrey Naughton	29
5	Divesh Srivastava	28
6	Michael Stonebraker	28
7	Joseph Hellerstein	27
8	Hector Garcia-Molina	26
9	Raghu Ramakrishnan	26

Table 1: Authors with more than 25 papers in ACM SIGMOD until 2007.

3.2 Incremental updates: An example

In the rest we focus on L_2 normalized indexes, since L_0 is easier to maintain. Consider Table 1 containing the names of authors who have more than 25 papers in ACM SIGMOD. In the following analyses and examples we will assume that all strings are decomposed into n-grams, and specifically view the tokens in $\mathcal{U}$ as n-grams. For example string "Michael Carrey", decomposed into 3-grams, becomes: {'##M', '#MI', 'MIC', . . ., ' CA', 'CAR', 'ARR', 'RRE', 'REY', 'EY#', 'Y##'}. The table contains 153 distinct 3-grams (including the special beginning and end of word 3-grams and after ignoring case). Only fourteen of them appear in more than one string. The most frequent 3-grams are 'CHA' and 'N##', with three appearances. Consider now that we build the inverted lists corresponding to the 153 3-grams, and that insertions, deletions and modifications arrive at regular time intervals. A single insertion or deletion of a string changes the total number of strings N in the table, and hence theoretically the idfs of all 3-grams, according to Equations (1) and (2). Complete propagation of the update would require recomputation of the length of each string (see Equation (5)), and hence updating all 153 lists. Consider now a modification of a single string, for example fixing the spelling mistake "Michael Carrey" to "Michael Carey". This modification has three consequences: 1. A change in the length of string 1; 2. The deletion (and subsequent disappearance) of 3-grams 'ARR' and 'RRE'; 3. The insertion of 3-gram 'ARE', which is a new 3-gram. A by-product of consequence 1 is that the partial weight of string 1 has to be updated in all inverted lists corresponding to the fifteen 3-grams comprising string 1. Nevertheless, consequences 2 and 3 have minor effects. The idfs of existing 3-grams do not get affected, since the deleted 3-grams were contained only in the modified string, and the newly inserted 3-gram did not exist in the table before the insertion. Alternatively, consider the modification "Michael Carrey" to "Micael Carrey", essentially deleting 3-grams 'ICH', 'CHA', and 'HAE'. The by-product of deleting one occurrence of 3-gram 'CHA', and hence changing the idf of this 3-gram, is that the lengths of all three strings containing this 3-gram change. This in turn means that the 46 lists corresponding

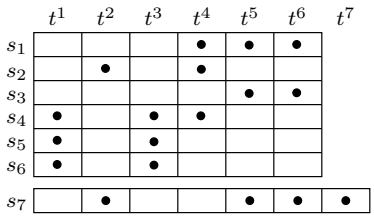

Figure 3: The inverted index represented conceptually by a sparse matrix M (with the actual ordering of strings within each list not being portrayed).

to the 3-grams contained in these three strings need to be updated, since they contain partial weights computed using the old lengths of the strings. Propagating an update that changes the idf of a very frequent 3-gram necessitates updating a large fraction of the inverted lists. Clearly, in order to be able to support incremental updates we need to propagate updates more efficiently, by relaxing the exact recomputation of idfs, but at the same time being able to deterministically compute query results with strict guarantees on the quality of answers.

3.3 Detailed analysis

First, we analyze the effects of fully propagating insertions, deletions and modifications to the inverted index. Then, we show how to propagate the updates in stages, in order to reduce the update cost in a way that limits the maximum loss in precision, when compared to a fully updated index.

For ease of exposition, consider that the inverted index is represented conceptually by a sparse matrix M where each row corresponds to a string in the database, and each column to a token from $\mathcal{U}$. Each cell $M_{i,j}$ contains a partial weight $w(s_i, t^j)$ if string s_i contains token t^j, and zero otherwise. An example is shown in Figure 3 (partial weights omitted for clarity).

3.3.1 Insertions.

Consider the example in Figure 3. An insertion can have one or more of the following consequences:

1. It can generate new tokens, and thus the creation of new columns in M. For example token t^7, after insertion of string s_7.

2. It might require adding new strings in existing inverted lists (as for columns t^2, t^5, t^6), hence affecting the idfs of existing tokens.

3. Most importantly, after an insertion the total number of strings N increases by one. As a result the idf of every single token gets slightly affected, which affects the length of every string and hence all partial weights in matrix M. A fully propagated update would have to touch every single non-empty cell in M.

4. String entries in inverted lists that have no connection to the directly updated tokens might need to be updated. This happens when the length of a string changes due to an updated token, triggering an update to all other lists corresponding to the rest of the tokens contained in that string.

5. The order of strings in a particular inverted list can change. This happens when a different number of tokens between two strings gets updated (e.g., three tokens in one string and only one token in another), hence affecting the length of one string more than the length of the other.

Let us concentrate on inverted list t^1, containing strings s_4, s_5, s_6. None of these strings contain any of the inserted tokens t^2, t^5, t^6, t^7. No matter how the lengths of these strings change (due to a change in N), the relative order of their partial weights will remain unaffected. Hence, updating the list requires only updating the partial weights of strings contained therein. Focus now on column t^4, containing strings s_1, s_2, s_4. Notice that $s_1 = \{t^4, t^5, t^6\}$ and $s_2 = \{t^2, t^4\}$. Clearly, depending on the new idf of t^2, t^5, t^6, the relative order of the partial weights of these two strings might actually change. The important thing to notice in this example is that an insertion can affect the ordering of strings even in lists that are not directly related to the update. Order changes are particularly expensive. If, for example, inverted lists are maintained as B-trees with partial weights as the key, simply modifying partial weights can be accomplished with a sequential scan of the leaf level and a bottom-up rebuild of the B-tree. Changing ordering, on the other hand, requires in addition re-sorting the data first. Notice also that identifying the lists containing a particular string whose partial weight needs to be updated is an expensive operation. To accomplish this we need to retrieve the actual string and find the tokens it is composed of. There are two alternatives for retrieving the strings. First, we can store the exact string along with every partial weight in all lists. This solution of course will duplicate each string as many times as the number of tokens it is composed of. The second option is to store unique string identifiers in the lists, and perform random accesses to the database to retrieve the strings. This solution will be very expensive if the total number of strings contained in a modified list is too large.

3.3.2 Deletions.

A deletion has the opposite effects of an insertion. A token might disappear if the last string containing the token gets deleted. Various entries might have to be deleted from a number of inverted lists, thus changing the idfs of existing tokens. The number of strings N will decrease by one. Thus, the idf of all tokens, and hence, the lengths and partial weights of all strings will slightly change, causing a cascading effect similar to the one described for insertions.

3.3.3 Modifications.

A modification does not change the total number of strings N, and hence does not affect the idf of tokens not contained in the strings being updated. Nevertheless due to a modification, new tokens can be created and old tokens can disappear. In addition, a modification can change the idf of existing tokens, with similar cascading effects. Going back to the example in Figure 3, assume that string s_7 is modified by a deletion of token t^2. The idf of t^2 changes and hence the lengths of all strings contained in list t^2 change, i.e. strings s_2, s_7. This means that we need to modify the partial weights of these strings in every other inverted list that contains them, i.e., lists t^4, t^5, t^6, t^7 in the example.

3.4 Relaxed propagation

Fully propagating updates for L_2 normalization is infeasible for large datasets if updates arrive frequently. The alternative is to relax equations (1) – (4) in order to limit the cascading effect of a given update. We can partially propagate updates in a way that provides guarantees on the quality of query answers.

3.4.1 Relaxation of N.

What causes the need for a complete recomputation of the index during insertions and deletions is the change of token idfs due to the modification of the total number of strings N. We can introduce a slack in how often we update N and keep idfs constant within a range of updates. Let N_b be the total number of strings when the inverted index was built. Then, N_b was used for computing the idfs of all tokens. Assume that we do not require a recomputation of idfs, unless if the current value of N diverges significantly from N_b (we quantify this change in the next section). Given a query q we need to quantify the loss of precision in evaluating the relaxed similarity $\mathcal{S}_2^{\sim}(q,s)$ for all $s \in D$, given idfs computed using N_b instead of N. Intuitively, we do not expect the idfs to be affected substantially given the log factor in Equations (1) and (2) for reasonable divergence of N. Relaxing N alleviates the need to recompute all idfs on a regular basis. Nevertheless, whenever N deviates outside some predefined bounds, we will have to rebuild the inverted index from scratch. The hope is that for balanced insertions and deletions, this will rarely occur.

3.4.2 Relaxation of df.

Assume now that we would also like to limit the effect of an update when the idf of a specific token changes, due to an insertion, deletion, or modification. Remember that a single token idf modification can have a dire cascading effect on a large number of inverted lists, as already discussed in Section 3.3.3. Assume that the idf of token t^i has been computed using document frequency $df_p(t^i)$ at the time the inverted index was built or the last time updates were propagated to the index. In this case we allow some slack in the recomputation of the exact idf of t^i by allowing the current document frequency $df(t^i)$ to vary within some predefined range, before actually updating $idf(t^i)$. First, notice that the effect of a small number of updates to a particular token is insignificant due to the log factor in Equations (1) and (2). In addition, the hope is to amortize the cost of propagating changes of frequently updated tokens. We analyze the exact loss in precision in Section 4.

Notice that the most severe cascading effects during updates are caused by the most frequent tokens, i.e., the tokens with large document frequency $df(t^i)$, and hence low inverse document frequency $idf(t^i)$. The most frequent tokens are obviously the ones that have highly populated inverted lists, and hence the ones causing the biggest changes to the inverted index during updates. In expectation, these will also be the tokens that will be updated more frequently. It is hence critical to limit changes to these tokens as much as possible. In our favor, notice that low idf tokens are actually the ones that contribute the least in similarity scores $\mathcal{S}_2(q,s)$, due to their small partial weights. Essentially, by delaying update propagation to low idf tokens, we limit the cost of updates significantly, and at the same time marginally affect query answer precision.

4. ANALYSIS OF LOSS IN PRECISION

In this section we analyze the exact loss in precision from delayed propagation theoretically. Let N_p, $df_p(t^i)$, $idf_p(t^i)$ be the total number of strings, the document frequencies, and the inverse document frequencies of tokens in $\mathcal{U}$ at the time the inverted index is built or the last time updates were propagated. Let N, $df(t^i)$ and $idf(t^i)$ be the current, exact values of the same quantities. Given a fully updated inverted index and a query q, let the exact similarity score between q and any $s \in D$ be $\mathcal{S}_2(q,s)$. Assuming now delayed propagation, let the approximate similarity score computed using quantities $\star_p$ be $\mathcal{S}_2^{\sim}(q,s)$. Our goal is to precisely quantify the relation between $\mathcal{S}_2$ and $\mathcal{S}_2^{\sim}$.

To simplify our analysis assume that the total possible divergence in the idf of t^i, by considering the divergence in both N and $df(t^i)$, is given by:

$$\frac{idf_p(t^i)}{\rho} \leq idf(t^i) \leq \rho \cdot idf_p(t^i), \tag{6}$$

for some value ρ. We will analyze the loss of precision with respect to ρ, and in doing so, we don't have to worry about actual changes in N or whether all idfs have been computed using the same N value or not, as long as all idfs are within the relaxation bounds. Notice that our analysis is independent of the particular form of the idf Equations (1) and (2), and will also hold for all other idf alternatives.

Consider query q and arbitrary string $s \in D$. Their cosine similarity is equal to

$$\mathcal{S}_2(q,s) = \frac{\sum_{t^i \in q \cap s} idf(t^i)^2}{\sqrt{\sum_{t^i \in s} idf(t^i)^2}\sqrt{\sum_{t^i \in q} idf(t^i)^2}}.$$

Let $x = \sum_{t^i \in q \cap s} idf(t^i)^2$ be the contribution of the tokens common to both q and s to the score. Let $y = \sum_{t^i \in s \setminus (q \cap s)} idf(t^i)^2$ be the contribution of tokens in s that do not appear in q, and $z = \sum_{t^i \in q \setminus (q \cap s)} idf(t^i)^2$ the contributions of tokens in q that do not appear in s. Thus,

$$\mathcal{S}_2 = f(x,y,z) = \frac{x}{\sqrt{x+y}\sqrt{x+z}}. \tag{7}$$

Notice that if $q = s$, then $y = z = 0$. Our derivation will be based on the fact that function (7) is monotone increasing in x, and monotone decreasing in y, z, for positive x, y, z. It is easy to see that the latter holds. We prove the former.

LEMMA 1. *$f(x,y,z) = \frac{x}{\sqrt{x+y}\sqrt{x+z}}$ is monotone increasing in x, for positive x, y, z.*

PROOF. Consider the function $g(x,y,z) = 1/f(x,y,z)^2$. $f(x,y,z)$ is monotone increasing in x iff $g(x,y,z)$ is monotone decreasing.

$$g(x,y,z) = \frac{(x+y)(x+z)}{x^2} = 1 + \frac{y+z}{x} + \frac{yz}{x^2}.$$

Since $1/x$ and $1/x^2$ are monotone decreasing in x, $g(x,y,z)$ is monotone decreasing in x, hence $f(x,y,z)$ is monotone increasing in x. □

Given the definition of x, y, z and relaxation factor ρ, it holds that:

$$\begin{aligned} x_p/\rho^2 \leq x_c \leq \rho^2 \cdot x_p \\ y_p/\rho^2 \leq y_c \leq \rho^2 \cdot y_p \\ z_p/\rho^2 \leq z_c \leq \rho^2 \cdot z_p, \end{aligned} \tag{8}$$

where x_p, y_p, z_p are with respect to propagated idfs, and x_c, y_c, z_c are the current, exact values of the same quantities.

We are given an inverted index built using idfs $idf_p(t^i)$, and a query q with threshold τ. We need to retrieve all strings $s \in D : \mathcal{S}_2(q,s) \geq \tau$. What is a threshold $\tau' < \tau$ s.t. retrieving all $s \in D : \mathcal{S}_2^{\sim}(q,s) \geq \tau'$ guarantees no false dismissals? Notice that for any s, given Lemma 1, the current score $\mathcal{S}_2(q,s)$ can be either larger or smaller than $\mathcal{S}_2^{\sim}(q,s)$, depending on which tokens in x, y, z have been affected. If $\exists s : \mathcal{S}_2^{\sim}(q,s) < \mathcal{S}_2(q,s)$, we need to introduce threshold $\tau' < \tau$ to avoid false dismissals. Hence:

$$\tau \leq \mathcal{S}_2 \leq \frac{\rho^2 x_p}{\sqrt{x_p/\rho^2 + y_p/\rho^2}\sqrt{x_p/\rho^2 + z_p/\rho^2}} = \rho^4 \mathcal{S}_2^{\sim} \quad (9)$$
$$\Rightarrow \tau' = \tau/\rho^4.$$

Clearly this bound is very loose and may introduce a very large number of false positives in practice, even for small slack ρ.

We consider now a more involved analysis that shows that given a relaxation factor ρ the actual loss in precision is a much tighter function of ρ. We want to quantify the divergence of $\mathcal{S}_2^{\sim}$ from $\mathcal{S}_2$, constrained on inequalities (8) and $\mathcal{S}_2(q,s) \geq \tau$. The query can be formulated as a constraint optimization problem. Minimize $f(x_p, y_p, z_p)$ constrained upon:

$$\begin{aligned} f(x_c, y_c, z_c) &\geq \tau \quad (10)\\ x_c/\rho^2 \leq x_p &\leq \rho^2 \cdot x_c \\ y_c/\rho^2 \leq y_p &\leq \rho^2 \cdot y_c \\ z_c/\rho^2 \leq z_p &\leq \rho^2 \cdot z_c, \end{aligned}$$

where inequalities (8) have been re-written after solving for x_p, y_p, z_p, instead of x_c, y_c, z_c (the exact same inequalities actually result in this case).

THEOREM 2. *$\tau' = \frac{\tau}{\tau + \rho^4(1-\tau)}$ is the minimum value of $f(x_p, y_p, z_p)$ that satisfies all of the constraints in (10).*

PROOF. First we show that $f(x,y,z)$ is minimized for $y = z$. Let $v = (y-z)/2$ and $w = (y+z)/2$. $f(x,y,z)$ is minimized, when $g(x,y,z) = f^2(x,y,z)$ is minimized (for positive x, y, z):

$$g(x,y,z) = \frac{x^2}{(x+w)^2 - v^2}.$$

$g(x,y,z)$ is minimized when the denominator is maximized, i.e., when $v^2 = 0 \Rightarrow y = z$ (given that y, z are independent of each other and v, w are only a rotational transformation of y, z).

Now, $f(x_p, y_p, y_p)$ is further minimized when x_p is minimized and y_p (or z_p) is maximized, according to Lemma 1. Hence, $f(x_p, y_p, z_p)$ is minimized at:

$$f(x_c/\rho^2, \rho^2 y_c, \rho^2 y_c) = \frac{x_c}{x_c + \rho^4 y_c}. \quad (11)$$

Consequently:

$$\begin{aligned} f(x_c, y_c, z_c) \geq \tau &\Rightarrow \quad (12)\\ \frac{x_c}{x_c + y_c} \geq \tau &\Rightarrow \\ y_c \leq x_c \frac{1-\tau}{\tau}&. \end{aligned}$$

Substituting Equation (12) into (11) we get:

$$\begin{aligned} f(x_c/\rho^2, \rho^2 y_c, \rho^2 y_c) &\geq \frac{x_c}{x_c + x_c \rho^4 \frac{1-\tau}{\tau}} \quad (13)\\ &= \frac{\tau}{\tau + \rho^4(1-\tau)}. \end{aligned}$$

Thus:

$$\tau' = \frac{\tau}{\tau + \rho^4(1-\tau)}, \quad (14)$$

satisfies all constraints. □

It is not hard to see that Equation (14) is always a tighter bound than (9) for all values of τ and ρ. Hence it is expected to yield significantly fewer false positives in most practical cases, while guaranteeing no false dismissals. To see this, consider the following example. For $\rho = 1.1$ and a query threshold $\tau = 0.9$, we need to lower the threshold to $\tau' = 0.86$, according to Equation (14), to guarantee no false dismissals; a very small decrease which in practice is expected to yield a small number of false positives. In contrast, by using the loose analysis and Equation (9), we would have to reduce the threshold to $\tau' = 0.61$ to achieve the same guarantee; a significant threshold decrease.

4.1 Practical considerations

We discuss here some practical considerations with respect to the lower bound presented in the previous section. Notice that our analysis assumed for simplicity the worst case scenario, where all token idfs take either their smallest or largest possible value. In practice, of course, the extreme values might not have been reached for all tokens. Notice that at query evaluation time we know the exact deviation of every token's propagated idf from its correct value. Clearly, we can take this information into account to limit false positives even further. We propose the following. First, we maintain the global maximum deviation $\sigma \leq \rho$ among all token idfs in $\mathcal{U}$. Then, at query time we compute the maximum deviation $\lambda \leq \rho$ among all token idfs in q. In deriving a lower bound for threshold τ', we use λ as a relaxation factor for x_p, z_p (the tokens in $q \cap s$ and $q \setminus (q \cap s)$), and σ for y_p (the tokens in $s \setminus (q \cap s)$). This lower bound in practice will be tighter than (14).

Finally, we discuss the practical meaning of relaxation factor ρ. Assuming modifications only (and hence a fixed value N) the basic intuition is that by allowing the idfs to deviate from their computed values by no more than factor ρ, we are essentially allowing token dfs to increase or decrease by a certain amount. A compound increase or decrease of a token's df will result in the need to propagate updates. Nevertheless, for low idf tokens (with very large df values) the probability of a compounded increase or decrease in the order of several thousands, is very small. On the other hand, for high idf tokens (with very small df values), the probability of a compounded increase or decrease by a small number is much higher. This works in our favor, since we would like low idf tokens (which have very small contributions to similarity scores) to be updated very infrequently, and high idf tokens (which have a very important contribution to similarity scores) to be updated more frequently.

5. UPDATE PROPAGATION ALGORITHM

We focus our attention to engineering issues related with supporting update propagation. We have an inverted index consisting of one inverted list per token in $\mathcal{U}$, where every list is stored on secondary storage. List entries $\langle s, w(s, t^i)\rangle$ are stored in decreasing order of partial weights w. To support update propagation we will need to perform incremental updates on the sorted lists. Hence, we store each list as a B-tree sorted on w. At index construction time we choose slack ρ.

Assume that we buffer arriving updates and propagate them in batches at regular time intervals (e.g., every 5 minutes). Let the updates be given in a relational table consisting of: 1. The type of update (insertion, deletion, modification); 2. The new data in case of insertions; 3. The old data in case of deletions; 4. Both the old and new data in case of modifications. We also build an idf table consisting of: 1. A token t; 2. The propagated idf of the token $idf_p(t)$; 3. The current, exact frequency of the token $df(t)$. Before applying the batch update to the index, we load the idf table in main memory. In practice, the total number of tokens $|\mathcal{U}|$ for most datasets is fairly small. Hence, maintaining the idf table in main memory is inexpensive.

In order to update the inverted lists efficiently we need to localize updates to each list and execute them in batch, taking advantage of data locality and buffering. We would certainly like to take advantage of sequential I/Os if the B-trees are clustered on disk, over thrashing multiple B-trees one after the other. Notice that we are forced to process updates in order of arrival to avoid conflicts (e.g., deletions on data that has not been inserted yet). But, localizing updates on a per list basis implies that we should not process updates one-by-one or in order of arrival. To circumvent this pitfall we use a journaling approach. We maintain a journal of updates on a per list basis in main memory. We process the updates in the update table in order of arrival, and record the necessary modifications to the in memory journals. We flush all changes to the corresponding B-trees all at once in the end, after performing certain allowable reorderings to improve performance. If any particular journal is becoming too large to fit in main memory, we flush the changes to the corresponding B-tree and clear the journal. After all journals have been populated, we process each journal in sequence, applying all the required changes to the respective B-trees. Notice that the total memory required to store the journals is linear to the size of the update table.

B-trees use the partial weight of an entry as the key. This implies that first, in order to delete an entry, we have to compute its original partial weight, and thus the length of the respective string in order to locate the entry in the B-tree. Furthermore, in order to modify an entry whose partial weight (and hence its length) has changed, we need to compute both the old and the new length of the respective string. For that purpose, the update table has to contain the old data for deletions and modifications. A modification of a key in the B-tree essentially translates into a deletion followed by an insertion.

While populating the journals we also maintain the main memory idf table. If a token is inserted we increase its exact document frequency by one. If it is deleted we decrease it by one. Tokens with document frequency zero are removed from the table. New tokens, that did not appear before, are inserted in the table and their current exact idf is computed. After the updates have been propagated to the B-trees we need to assess whether any token idfs have deviated more than the allowed slack ρ in any direction. We scan the idf table and identify the corresponding tokens and compute their idfs.

Algorithm 5.1: PROPAGATE(D, U)

```
Let hash table H containing token df, idf_p
Let M ← ∅, Journals J, N
for each {op, s} ∈ U
  if op is an insertion
    N+ = 1
    for each t ∈ s
      do { if t is a new token
           do { Insert in H
                df(t) = 1
                idf_p(t) = log_2 (1 + N/df(t))
           else df(t)+ = 1
    Compute L_2^p(s)
    for each t ∈ s : Add insertion ⟨s, w_p(s,t)⟩ to J(t)
  if op is a deletion
    N− = 1
    Compute L_2^p(s)
    for each t ∈ s :
      do { df(t)− = 1
           Add deletion ⟨s, w_p(s,t)⟩ to J(t)
Commit journals
for each t ∈ H
  if df(t) = 0 : Remove t from H
  if NOT idf_p(t)/ρ ≤ log_2 (1 + N/df(t)) ≤ idf_p(t)ρ
    do { M ← t
         idf(t) = log_2 (1 + N/df(t))
for each t ∈ M
  for each s ∈ inverted list t
    do { Compute new L_2(s)
         for each t' ∈ s \ M
           do {Add modification ⟨s, w(s,t')⟩ to J(t')
  Rebuild inverted list t
Commit journals
```

Figure 4: Update propagation algorithm.

Assuming that multiple token idfs have changed, we need to scan the B-trees corresponding to these tokens, and retrieve all strings contained therein (which requires at least one random I/O per string id for retrieving the actual strings from the database). Then, we compute the new lengths of the strings, given the updated token idfs. Finally, first we rebuild the B-trees corresponding to tokens whose idf has changed, and also update all other B-trees that contain those strings. Every time we process a string we store the string id in a hash table and make sure that we do not process that string again, if it is subsequently encountered in another B-tree (this will be the case for strings that contain multiple tokens whose idfs have changed). Pseudocode of the update propagation algorithm is shown in Algorithm 5.1.

6. EXPERIMENTS

We implemented a full featured prototype to test our update propagation framework with real datasets. The prototype is implemented in C++ and we used the open source Oracle BerkeleyDB [19] as the underlying DBMS for the B-tree indexes. For our experiments we used the publicly available DBLP citation database [17], and the Business Listing (BL) database from a major online website (YellowPages). All experiments were run on a two four core Intel(R)

(minutes)	2-grams		3-grams		4-grams	
	S1	S3	S1	S3	S1	S3
Process strings	0.98	0.98	1.21	1.23	1.56	1.53
Sort n-grams	1.58	4.96	6.51	5.61	63.7	6.58
Create B-trees	4.8	3	7.97	2.96	69.3	16.8
Total	7.36	8.94	15.7	9.81	134.6	24.91

Table 2: Construction cost for DBLP.

Xeon(R) CPU 2.66 GHz, with 16 GB of main memory. We focused on a DBLP table containing associations between author names and publication ids for books, articles and proceedings. We downloaded daily snapshots over a 30 day period (February 11th, 2008 to March 11th, 2008) and computed diffs, which we then used to create update tables. The original table contains 2460433 author/id pairs, 5712041 total words and 269281 distinct words. The average author name size is 13 characters, and there are 2050 distinct 2-grams, 23666 3-grams, and 161213 4-grams in the original table. There were 33461 total updates produced within a month (approximately 1000 updates per day). These consisted of 32121 insertions and 1340 deletions. In reality, deletions correspond to modifications of existing records, which we represent as deletions followed by insertions for simplicity; there were no actual deletions from the DBLP database. We also run experiments using the BL database. New business listings are added on a daily basis, hence every listing is associated with an insertion timestamp. We select 15 million entries as the dataset, and another 30 thousand entries as insertions. The dataset contained approximately 52 million words and 633 thousand distinct words. The average business name length is 20 characters. There are 37524 distinct 3-grams.

6.1 Construction cost

First, we build the inverted index on the final DBLP and BL tables (after incorporating all updates). There are three strategies we can use to generate n-grams: S1. Selection sort and subsequent bulk-loading of B-trees; S2. Direct insertion into B-trees; S3. External sort and subsequent bulk-loading of B-trees. We use 4096 byte page sizes for the B-trees in all cases, and 2-grams, 3-grams, 4-grams for this experiment. We use 300MB as an aggregate file buffer for S1, S2 and as a main memory buffer for external sorting. We omit results for S2 since it is not competitive. The time to run each step of the building process is shown in Table 2 and Figure 5. The construction cost increases super-linearly to the number of n-grams (due to the sorting step), and the number of n-grams increases for larger n, making construction more expensive (from 40 million 3-grams to 42 million 4-grams, and from 23K to 161K lists). The most expensive step is the creation of B-trees, but as the number of n-grams increases, the cost of sorting n-grams approaches that of creating B-trees. Notice the overwhelming cost of managing buffered files for S1 (both for writing during sorting and for reading during B-tree creation) versus the comparative times for S3. From the construction cost of the BL dataset we can see that for large datasets strategy S1 performs as well as S3 (for this dataset the n-gram generation process for 15M strings creates approximately 365 million 3-grams, versus the 40 million 3-grams for the DBLP table). Increasing the dataset size even further will exacerbate the difference between S1 and S3, as the number of n-grams increases linearly but the number of lists is almost constant.

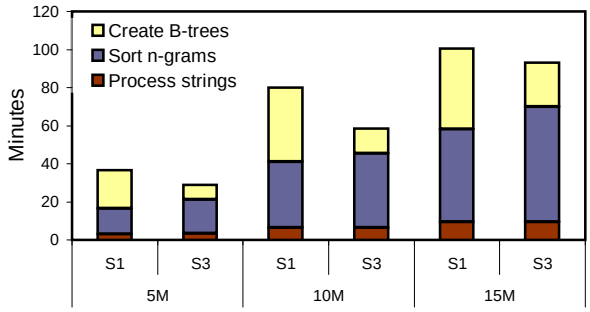

Figure 5: Construction cost for BL.

Next, we compare the construction cost of using L_0 normalization. L_0 normalized lists require only one pass of the data to compute idfs and lengths. Nevertheless, sorting the n-grams and creating B-trees cannot be avoided. Given that the string processing step is the most inexpensive (7 to 44 times cheaper than B-tree creation in practice) it is clear that L_0 normalization does not offer any significant advantages in terms of index construction (it is only marginally faster than L_2 construction). At the same time it does not offer the benefits of L_2 normalization in terms of query efficiency (see Section 2). We will also show in the next section that L_0 normalization is not significantly faster than the proposed lazy update propagation for L_2 normalization in terms of incremental update cost.

Given that the cost of extracting the n-grams and creating B-trees increases super-linearly to the total number of strings in the base table, it is clear that for tables with tens of millions of entries, even for relatively short strings, the cost of rebuilding the inverted index quickly becomes infeasible on a daily basis. It should be stressed here that we could avoid building sorted inverted lists and hence having to maintain B-trees, but that would result in slower query execution times based on merging strategies that need to exhaustively read lists. We pay the cost of slower update times for faster query execution.

6.2 Update propagation cost

Next, we construct the inverted index on the original DBLP table (before incorporating the updates), and measure the cost of subsequently updating the index using our propagation framework. Our goal is to be able to apply the updates in a semi-real-time fashion, where we buffer updates and propagate in regular time intervals (e.g., every 5 minutes). The hope is that propagating even the 1000 *daily updates* will require significantly less time than 5 minutes. In the rest we concentrate mostly on the DBLP data which is publicly available, since results for the BL dataset followed similar trends.

Figure 6 shows the time it takes to propagate a batch of updates for various batch sizes using our algorithm. For this experiment we use a relaxation factor of 3%. First, we report the time it takes to parse one batch of updates, extract the 3-grams, retrieve their idfs and populate the journals with the appropriate changes for each B-tree. Next, we report the time it takes to commit these changes to the B-trees. Finally, we report the time it takes to commit all changes related to idfs that have exceed their relaxation bounds. The most important observation here is that such changes are indeed very small and affect only an extremely small fraction of the strings. The bulk of the cost is incurred by the B-tree

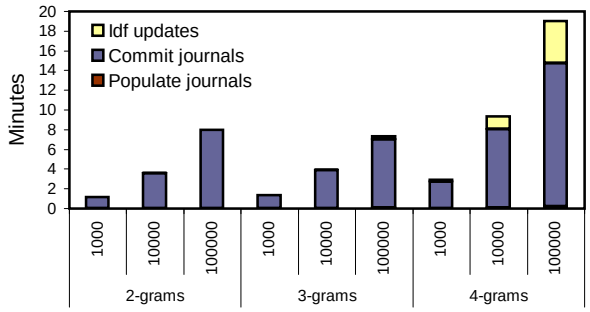

Figure 6: Cost of propagating one batch of updates for various batch sizes (DBLP; relaxation 3%).

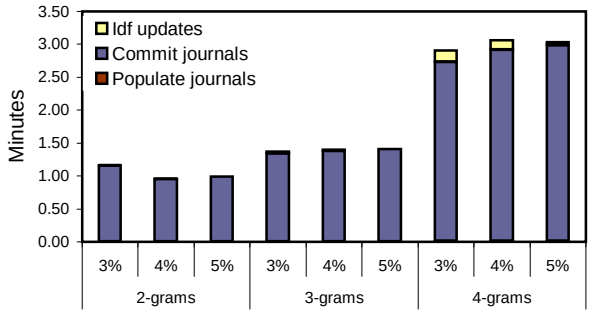

Figure 7: Cost of propagating updates for various relaxation factors (DBLP; batch size 1000).

	Batch Size		
Statistic	1000	10000	100000
# of update operations	539767	539767	539767
# of out of bounds idfs	95	88	83
# of affected ids	252	230	221
# of affected lists	2138	1715	1321
# idf related updates	4216	3744	3544

Table 3: Statistics of update propagation for various batch sizes (DBLP; 3-grams; 3%).

operations, that cannot be avoided if we want the updates to be reflected in the index immediately, irrespective of the normalization used.

The cost of incremental updating using L_0 normalization consists of the first two steps (populating and committing the journals, and hence is only marginally faster than our relaxed propagation policy. In other words, we are able to provide the full benefits of using L_2 normalization, for a small penalty in update cost. In particular, for the 1000 batch size there are a total of 34 batches. The average time to propagate a single batch when using 3-grams is 1.37 minutes for L_2 and 1.34 minutes for L_0. Similarly, for a 10000 update batch it takes on average 3.97 minutes for L_2 and 3.88 minutes for L_0, and for the full 33461 updates 7.35 minutes for L_2 and 7.03 minutes for L_0. Clearly, we are able to process more than 10000 updates on a per 5 minute update window.

An indication of the cost of incremental updates is the total number of update operations incurred. Table 3 lists several statistics (here we use relaxation 3% and 3-grams). We list: 1. The total number of update operations solely from executing insertions and deletions of strings; 2. The number of idfs that fall out of bounds during the update operation; 3. The number of string ids contained in the respective inverted lists whose new lengths will have to be propagated; 4. The number of lists that will be affected by this propagation operation; 5. The total number of update operations due to idf changes. The total number of list updates incurred from idf propagation is two orders of magnitude smaller than the total number of normal update operations. This verifies our intuition that propagating updates for infrequent n-grams only will have a very small impact in the overall cost. Notice that as the batch size increases the total number of idfs that falls out of bounds decreases since insertions and deletions average out in the end, before the updates need to be propagated by committing the journals.

Figure 7 shows the effects of varying ρ on update performance. We use 1000 updates per batch for this experiment.

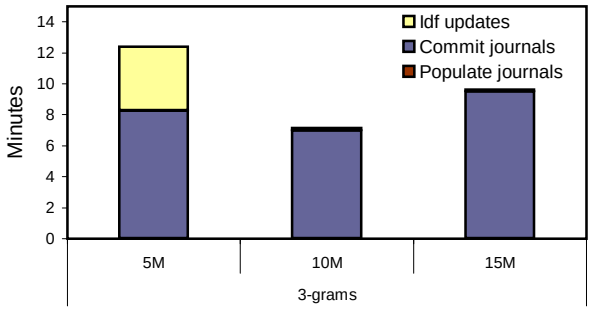

Figure 8: Cost of propagating updates for various dataset sizes (BL; relaxation 4%).

We can clearly see from this plot that even for tight relaxation factors the cost of propagating updates is insignificant in contrast to the cost of the update itself. Nevertheless, we can see that we are able to commit each 1000 batch within 3 minutes when using 2-grams, 3-grams, or 4-grams.

Figure 8 shows the results of a scale-up experiment using the BL dataset. We create an initial index using 5, 10, and 15 million entries, and then apply the 30 thousand updates in order to measure the update performance as a function of the underlying index size. We use 3-grams and 4% relaxation. We observe that the cost of updating B-trees due to idf changes is very small for large datasets, but significant as the dataset size becomes smaller. This is due to the fact that the 30 thousand updates become a significant percentage of the total number of strings N_b as the dataset size decreases, meaning that a larger percentage of token idfs is expected to exceed the relaxation bounds. Notice also that the time it takes to update the B-trees doubles with respect to the DBLP dataset (probably due to the longer string lengths of the updates performed). The most important observation though is that the cost of updating the B-trees remains almost constant across all dataset sizes within BL. This indicates that propagating updates scales extremely well with respect to the size of the underlying index, which is expected since that cost is directly proportional to the size of the update table and not the size of the index (a small increase of this cost is expected for larger datasets as the underlying B-trees become larger and more expensive to maintain per update operation, but only with a logarithmic factor that depends on the fanout of the trees).

Finally, for completeness Figure 9 plots the update propagation cost for the 5M BL data as a function of ρ, for the 10000 batch size. The cost of update propagation due to idf changes decreases as the relaxation factor increases.

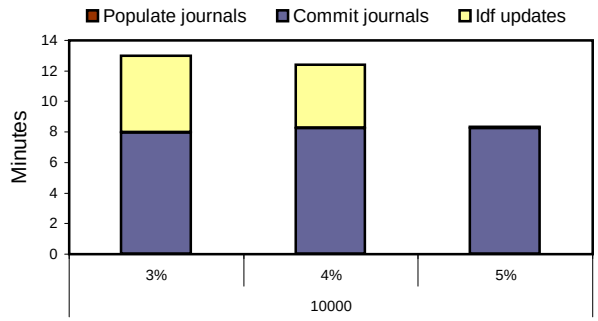

Figure 9: Cost of propagating updates for various relaxation factors (BL; 5M).

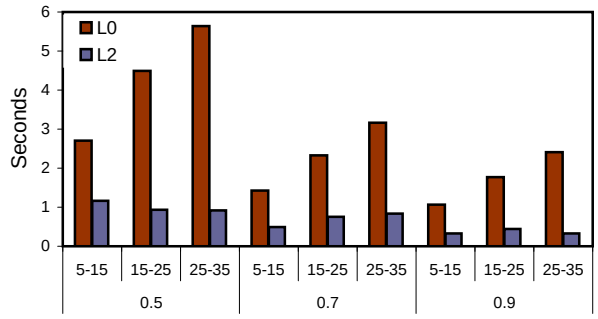

Figure 10: Average query cost for retrieving the query results (DBLP).

6.3 Query accuracy

A qualitative analysis of query results for a variety of normalization techniques can be found in [5]. In this section we evaluate the difference between L_0 and L_2 normalization in terms of speed, as well as the loss in accuracy resulting from our proposed lazy update propagation framework. We measure Recall, Precision and Average Relative Error (ARE) of scores. Recall is measured as the percentage of true answers retrieved by the lazy updated index, over the total number of answers retrieved by a fully updated index. Given our theoretical analysis, we expect 100% recall in all cases. Precision is measured as the number of true answers retrieved, over the total answers retrieved. This is a measure of the number of false positives returned. Finally, we also measure the average relative error in the scores computed by the inverted index with lazy propagation, when compared to a fully updated index. For this set of experiments we randomly picked 100 strings out of the 32121 newly inserted strings as queries. We report averages in the graphs.

Figure 10 plots the average time required to retrieve answers for various thresholds and query lengths (for L_0 we retrieve the top-k results that correspond to the answer given by L_2). We created 3 query sets containing 100 strings each, all within the respective string length bounds. We can observe that indeed L_2 normalized indexes are at least twice as fast and up to one order of magnitude faster, in all cases.

Table 4 plots average recall, average precision and average relative error as a function of ρ. For this experiment we use a query threshold $\tau = 0.7$, yielding an average of 22 true answers per query. As expected our framework has 100% recall. Clearly, using larger ρ in our experiments has no negative effect in terms of query accuracy. Notice the very small number of false positives, even for 5% relaxation factor. Remember that for 5% relaxation, the update propagation cost was insignificant (refer to Figure 7). This experiment verifies our intuition that propagating updates only for high idf n-grams will keep errors very low, while enabling very fast updates. Finally, it is important to note that the average relative error in the scores computed is close to zero.

	ρ (%)		
	3%	4%	5%
Recall	1	1	1
Precision	0.901	0.901	0.901
ARE	$3.4 \cdot 10^{-5}$	$3.4 \cdot 10^{-5}$	$3.4 \cdot 10^{-5}$

Table 4: Accuracy as a function of relaxation factor (DBLP).

	τ			
	0.5	0.6	0.7	0.8
Recall	1	1	1	1
Precision	0.728	0.79	0.901	0.974
ARE	0.0002	0.0001	$3.4 \cdot 10^{-5}$	$6.9 \cdot 10^{-6}$

Table 5: Accuracy as a function of query threshold (DBLP).

Finally, we present accuracy experiments as a function of query threshold. For this set we used 5% relaxation. Table 5 shows the results. The number of false positives increases as the query threshold decreases, first due to the larger exact number of answers per query, second due to the smaller reduced thresholds τ' given by Equation (14). For example, for 5% relaxation and $\tau = 0.5$, $\tau' = 0.47$. Clearly, if we are willing to tolerate the slightly increased propagation cost for smaller ρ, query precision will improve even for smaller thresholds. Still, 70% precision for such small thresholds is remarkable given the small cost we incur to keep the index updated. In most practical scenarios, thresholds vary between 0.8 – 1, in which case our framework yields above 90% precision.

7. RELATED WORK

String similarity operators based on a variety of similarity measures have been proposed in [2, 3, 4, 6, 14, 21]. Efficient indexes for these operators have been proposed in [2, 6, 12, 13, 21]. Most approaches build some form of inverted indexes on the tokens contained in the dataset. Some indexes are based on relational database technology (represented as relational tables and expressed as SQL queries). Others are specialized inverted lists stored on secondary storage. Due to the global nature of the weights associated with entries in length normalized indexes, updates are known to be expensive for all existing approaches. There has been very little work on incremental updates in this context. Previous work assumes buffering of updates and full recomputation of the indexes on regular time intervals.

Other approaches (e.g., Lucene [1]) choose to use simpler length normalization techniques (i.e., L_0) which makes incremental updates easier and marginally faster to handle. Nevertheless, the drawback is significantly slower list merging algorithms. Additionally, the unrestricted nature of similarity scores when simple normalization methods are used, limits the practicality of these measures in certain cases (it is not clear what the best similarity threshold for a given query should be, and the resulting scores cannot be intu-

itively compared). Generally speaking, previous approaches concentrate mostly on information retrieval on large documents, ignoring special properties that are true for queries on short strings.

The only work that has considered incremental update propagation for length normalized indexes is [15]. That work focused on `TF/IDF` cosine similarity and purely relational based indexes (the inverted index is in the form of a relational table and queries are expressed using SQL). It concentrated on heuristics, without providing a rigorous analysis of the loss in precision. The present work uses these ideas as a foundation for the new framework, but also provides strict lower bounds on the loss of precision. In addition, the present work describes in detail a framework based on highly efficient specialized inverted indexes stored on secondary storage, rather than using relational database technology. This is highly desirable for online applications where fast query response time is instrumental, and immediately reflecting the effect of updates in the system is essential.

8. CONCLUSION

We developed a framework for efficient incremental updates on inverted indexes for approximate string matching. We argued that L_2 length normalization has very appealing properties with respect to query evaluation efficiency and answer quality, and showed that it is also amenable to incremental update propagation with guarantees in the loss of precision. We analyzed the loss theoretically, described engineering issues in detail, and evaluated the efficiency of the proposed framework experimentally on a real datasets using a fully functional prototype. In the future we plan to investigate if probabilistic similarity measures are also amenable to lazy update propagation with similar guarantees.

9. REFERENCES

[1] Apache Lucene. http://lucene.apache.org/.

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

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

[5] 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.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[7] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16, 2007.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 102–113, 2001.

[10] FAST - Enterprise Search. http://www.fastsearch.com.

[11] Google. http://www.google.com.

[12] 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.

[13] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[14] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *Proc. of Very Large Data Bases (VLDB)*, pages 1078–1086, 2004.

[15] N. Koudas, A. Marathe, and D. Srivastava. Propagating updates in SPIDER. In *ICDE*, pages 1146–1153, 2007.

[16] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *Proc. of ACM Management of Data (SIGMOD)*, pages 802–803, 2006.

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

[18] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[19] Oracle. Oracle Berkeley DB. http://www.oracle.com/technology/products/ berkeley-db/db/ index.html.

[20] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau. Okapi at TREC. In *Text REtrieval Conference (TREC)*, pages 21–30, 1992.

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

[22] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proc. of ACM Management of Data (SIGMOD)*, pages 353–364, 2008.