LSH-Based Probabilistic Pruning of Inverted Indices for Sets and Ranked Lists*

Koninika Pal
TU of Kaiserslautern
Kaiserslautern, Germany
pal@cs.uni-kl.de

Sebastian Michel
TU of Kaiserslautern
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

We address the problem of index pruning without compromising the quality of ad-hoc similarity search among sets and ranked lists. We discuss three different ways to prune the index structure and, by linking the index structure with the concept of Locality Sensitive Hashing (LSH), we introduce two solutions to query processing over the pruned index. Through a probabilistic analysis we ensure that a user-defined recall goal is still guaranteed. We are able to formulate an optimization problem that can determine the optimal pruning factor for all three pruning methods. The experimental evaluations over real-world data validate that the optimal pruning factor indeed ensures the recall goal without any significant effect on the quality of similarity search on a much smaller index.

KEYWORDS

Similarity Search, Index Pruning, LSH

ACM Reference format:
Koninika Pal and Sebastian Michel. 2017. LSH-Based Probabilistic Pruning of Inverted Indices for Sets and Ranked Lists. In *Proceedings of WebDB'17, Chicago, IL, USA, May 14, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3068839.3068845

1 INTRODUCTION

Sets are ubiquitously used to represent an accumulation of properties associated with a specific object, for instance opinions expressed by users, tags assigned to tweets, or topics associated with news articles. Ranked lists, add on top of sets the possibility to specify a preference among the included properties/elements. This is specifically often observed for "Hall-of-Fame" style rankings that reflect user preferences or anti preferences on movies, celebrities, foods, cities, etc. Performing similarity search for such data means determining users that share the same or contrasting interests or opinions, finding tweets that have similar hashtags, or blog posts around specific topics. The rankings and sets motivated by the aforementioned scenarios are usually not very long, and it has been observed [3] that in such cases, inverted indices are an ideal solution to processing ad-hoc similarity queries.

*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

WebDB'17, Chicago, IL, USA

978-1-4503-4983-3/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3068839.3068845

Inverted indices are simple structures that map properties, or pairs/triplets of properties to sets/rankings that feature these properties. With one lookup, one can, for instance, determine all users that share the same interest in a topic 'soccer'. Augmented with sentiment, one can naturally also capture positive or negative feelings for a topic. This simplicity and general applicability renders inverted indices a widely known and used concept.

Essentially, inverted indices resemble a hash map where the keys are items or pairs etc. derived from the sets/rankings to be indexed. This works well for similarity search, since it efficiently allows determining sets/rankings that share common entries, a necessity for being really similar. Why this works well and how many keys need to be accessed from the index at query time can be understood by seeing the relatedness of inverted indices to locality sensitive hashing (LSH).

In LSH, hash functions are created that map objects to hash buckets with the desired property that similar objects have a higher chance to collide in the same bucket. In fact, this interpretation, tailored to specific distance measures, allows tuning query processing to outperform prefix filtering, since results can be found in much less index accesses, shown in our prior work [10].

Let us emphasize the connection between LSH and inverted indices through an example. Table 1 shows three sets of integers. The connection between inverted indices and locality sensitive hashing (LSH) is apparent: We can define a simple hash function family that projects sets τ_i to binary space based on presence or absence of the elements inside it, e.g., $\{h_x(\tau_i) = 1 | x \in \tau_i\}$. Hence, the bucket label '1' for function h_2 is similar to the key entry '2' in inverted index, both of them hold the sets where number '2' appears. In [6, 10], hash function families are discussed that are proven to be locality sensitive for Hamming distance and generalized Kendall's Tau distance, respectively.

Now, in this work, we focus on how to optimize the size consumption of an inverted index by exploiting LSH properties. We will, throughout the coming sections, focus on sets, but come back to handling ranked lists in Section 5, where we instantiate the proposed "theory" for explicit scenarios.

We consider the problem of similarity search over sets (which can be applied also to ranked lists) where the size of the queries is the same as for the stored sets. This is in contrast to common document retrieval, where queries are normally expressed in much fewer words than the stored documents contain. Hence, finding the results based on score-based document search vastly differs from the scenarios we consider in this work. Similarly, let us further briefly mention that there are many lossless compression methods and lossy static index pruning methods for Web Search engines and document retrieval in general [11, 12, 15]. We believe that lossless compression techniques are fully orthogonal to our approach, but any detailed investigations are beyond the scope of this first paper.

1.1 Problem Statement and Setup

We consider a data collection $\mathcal{S}$ comprising of sets τ_i. Each $\tau_i \in \mathcal{S}$ has a domain D_{τ_i} of items it contains. The global domain of items is then $D = \bigcup_{\tau_i \in \mathcal{S}} D_{\tau_i}$ and $|D| = n$. We assume all sets have the same size. Table 1 shows three sets holding five elements each.

At query time, a user provides a query set q, a distance threshold $\theta \in [0, 1]$, and a distance function d. Our objective is to determine all sets $\tau_i \in \mathcal{S}$ that have a distance less than or equal to θ, so the result $\mathcal{R}$ of a query can be written as

$$\mathcal{R} := \{\tau_i \mid d(\tau_i, q) \leq \theta, \tau_i \in \mathcal{S}\}$$

As mentioned above, we can build a simple inverted index over $\mathcal{S}$, to look up—at query time—those sets that have at least one element overlapping with the query's elements. This obviously avoids scanning through the sets, and, in fact, it is known [3] that inverted indices can handle set similarity queries very efficiently. Particularly, the distance to the query is evaluated only for the sets which hold overlapping elements with the query. Considering the example in Table 1, for a query $q = \{8, 7, 0, 6, 9\}$, the set τ_1 does not overlap at all with the query's elements, while τ_2 and τ_3 do overlap.

In this paper, we address the problem of rendering inverted indices more space-efficient by eliminating some of the stored posting lists, posting list entries, or both, controlled by a pruning parameter $\phi \in [0, 1]$.

If a query is executed over such a pruned index, the query returns a result set $\mathcal{R}_p$ that is a subset of the true result set $\mathcal{R}$ given above.

Our objective is to find the optimal pruning factor ϕ such that the results of the similarity search satisfy a user-defined recall level ϱ. We can formalize this task as follows:

$$\begin{aligned} &\textit{maximize} && \phi \\ &\text{subject to} && \mathcal{R}_p/\mathcal{R} \geq \varrho \end{aligned}$$

Note that we assume the distance threshold θ to be strictly smaller than the normalized maximum possible distance, hence, $0 \leq \theta < 1$. Therefore, the inverted index can find all result sets as all results need to have at least one overlapping item with the query.

1.2 Contributions and Outline

In this work, we make the following contributions:

- We discuss three ways to prune an inverted index and propose two different ways to query a pruned one.
- We propose a way to find the optimized maximum pruning factor ϕ^*, still ensuring a user-given recall goal.
- For this, we present a probabilistic analysis to find the number of index entries needed to be accessed to reach the predefined recall goal for the similarity search over pruned inverted indices.

The paper is organized as follows. Section 2 introduces preliminaries for query processing in LSH. Section 3 introduces the three, admittedly very straightforward, pruning techniques. The probabilistic analysis of query processing in pruned indices and optimization problem of finding the maximum pruning factor are discussed in Section 4. Section 5 presents two case studies of applying our techniques to inverted indices for sets and top-k rankings. A detailed experimental study is presented in Section 6. Section 7 presents related work, while Section 8 concludes the paper.

$\tau_1 = \{2, 5, 4, 3, 1\}$	$7 \rightarrow \langle\tau_2\rangle, \langle\tau_3\rangle$
$\tau_2 = \{1, 4, 7, 5, 2\}$	$5 \rightarrow \langle\tau_1\rangle, \langle\tau_2\rangle, \langle\tau_3\rangle$
$\tau_3 = \{0, 8, 7, 5, 6\}$	$4 \rightarrow \langle\tau_1\rangle, \langle\tau_2\rangle$
	...

Table 1: Sample sets (left) and inverted index (right)

2 PRELIMILARIES

An inverted index is a simple data structure that resembles a mapping between items and objects in the sense that it maps an item to all objects that contain this item. In our case, these objects are sets, or, as we will later see, ordered lists of items. In information retrieval, inverted indices are commonly used to map terms to documents that contain the term. Table 1 shows the partial inverted index for the example given earlier.

Query processing in an inverted index follows a simple **filter and validate procedure**. For a given query object, we first generate all the keys (e.g., single items or pairs) from the query and then, for each key, look up the index to retrieve candidate result objects. Finally, we calculate the true distance between the candidates and the query to find if they comply to the given distance threshold θ_d and, by doing so, determine the final result set $\mathcal{R}$.

More advanced methods, like the popular prefix filtering approach [13], derive a query-dependent bound on the number of keys that need to be looked-up. This bound is determined by finding the sufficient overlap between the objects for user-defined distance threshold. Ideally, this allows accessing only a small fraction of posting lists compared to the simple approach that accesses all possible keys from a query.

Unlike counting the overlapping elements, *if we can establish a one to one mapping between the entries of hash tables in LSH and the inverted index then by exploiting the collision probability of LSH we can also find the number of keys in index we need to look up to find the candidates of similarity search.*

LSH have the desired property that similar objects have higher probability to collide into the same bucket than the dissimilar ones. To decrease the probability that objects end up in the same bucket by chance, the outcome of multiple hash functions is used to identify a bucket. That means, if we use m hash functions jointly together, the probability that objects are mapped into the same bucket becomes P_1^m where P_1 is the collision probability of the individual hash functions. While this improves the precision of a bucket look up, the recall tends to suffer. To circumvent this, l such hash tables are created and, consequently, if an object is found in any of these l hash tables, it is considered a candidate for the final result which is determined by computing the distance to the query and returning the ones that satisfy the distance threshold θ. The downside of this elegant and tunable approach is that, as m and l increases, the space needed to store the LSH index also increases.

In this paper, we aim at rendering inverted indices more space efficient, by index pruning and adapted query processing methods, while still matching a given recall goal. Recall quantifies the fraction of actual query results returned.

In LSH with parameters m, l, and collision probability P_1, the probability of a candidate becoming a positive one (that is, a final result) is calculated by the following equation.

$$\varrho = 1 - (1 - P_1^m)^l \tag{1}$$

Here, by fixing the user-defined recall requirement ϱ, we can tune the parameter l, i.e., the number of keys that need to get looked-up in index at query time.

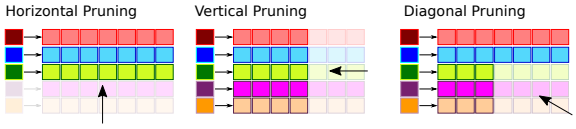

Figure 1: Illustration of three ways to prune an inv. index.

3 HORIZONTAL, VERTICAL, AND DIAGONAL INDEX PRUNING

Optimizing the space requirement of an inverted index is important as the index should be held in main memory, in order to avoid expensive random I/O when accessing buckets. As pointed out in the preliminaries above, in order to tune precision of lookups and recall of the obtained results, multiple hash tables are used where each hash function is a composition of m hash functions.

There are several methods in literature (cf., Section 7) that aim at increasing query efficiency by reducing the number of index accesses at query time. However, such methods do not have any potential for reducing the size of the index. In this section, we discuss three simple ways to drop parts of the inverted index structure, as illustrated in Figure 1.

3.1 Horizontal Pruning

In this approach, we randomly select a fraction ϕ of the total index keys and remove the corresponding posting lists completely. Considering the elements are uniformly distributed, such a random removal reduces ϕ fraction of space for storing posting lists.

On the other hand, instead of random removal, the most frequent or least frequent ϕ fraction of index keys can be dropped (thus, drop very long, or very short posting lists). In this case, we can also calculate the reduced space for holding posting list based on the skewness of the data. For example, having data following a Zipf distribution with skew $\nu = 1.1$ means that the top-20 entries cover 30% of total objects listed in all posting lists. Clearly, random removal of ϕ fraction of entries reduces less space than removal of top ϕ fraction of entries and more space than removal of bottom ϕ fraction of entries.

3.2 Vertical Pruning

Unlike removing index keys, thus, entire posting lists, in this strategy, we randomly drop a ϕ fraction of entries from each posting list. Randomly dropping elements from postings is independent from the distribution of the elements, thus, dropping a ϕ fraction in vertical pruning directly reduces space to store posting lists by a factor of $1 - \phi$. If we assume more information about the usefulness of the stored posting list entries, we can instead of randomly dropping entries drop the least important ones from each list. In information retrieval literature, this is known as term-based static index pruning and scores that could be useful to decide what to drop can be measures like tf-idf or KL-divergence.

3.3 Diagonal Pruning

In this strategy, we drop a fraction of ϕ elements from each of the original sets and then create an index based on the remaining elements. Hence, if a specific element is, by chance, removed from all the objects, the index entry corresponding to that removed element will not appear in the index which resembles the idea of vertical pruning. On the other hand, if a specific element is removed only from a few sets, then those sets will not appear in the posting list of that element, which resembles horizontal pruning.

As this method prunes the index both horizontally and vertically, we called it diagonal pruning. It generalizes the document-centric pruning method, known from the area of information retrieval, as less important terms are removed from each document before generating the index. We can observe here that if elements are uniformly distributed among sets, this pruning method tends to prune the index equally from vertically and horizontally "direction". On the other hand, if data has a skewed distribution, then this method tends to prune the index more horizontally rather than vertically.

4 QUERY PROCESSING ON PRUNED INDICES

As we drop index entries or objects from hash tables, there is a chance that a query object does not collide with a true result object for a particular hash function. This can happen if the entry where the hash function maps the query to was dropped at pruning time. Therefore, all pruning methods modify the actual collision probability of hash functions, which directly affects the query processing. Here, we will discuss how we can approximate the effect of pruned indices on query processing by calculating the modified collision probability for LSH.

4.1 Ad-hoc Query Processing

A properly set up LSH index assures that by issuing l accesses to the hash tables, a predefined recall goal is met. For a pruned index, this is not anymore the case. One first solution to overcome this problem is simply to use more than l hash functions to map the query into the pruned index, until we reach the required l successful accesses to buckets. In worst case, we might need to use a large number of hash functions that map the query to the index to achieve the required number of successful accesses. This approach does not include any prominent extra overhead in runtime performance, as runtime is mainly dominated by the validation of retrieved candidates found in successful bucket access.

4.2 Probabilistic Query Processing

Unlike the above ad-hoc approach, in probabilistic query processing, we modify the collision probability of LSH schemes based on the amount of index lists that are pruned and the way of the pruning (horizontal, vertical, or diagonal). Equation 1, that quantifies the expected recall, is adapted by introducing a factor f_Y, that scales the original collision probability P_1.

$$\varrho = 1 - (1 - f_Y \cdot P_1^m)^{l_Y} \tag{2}$$

We approximate the number of required accesses l_Y, where the subscript Y indicates the pruning method that was used, i.e., horizontal, vertical, or diagonal. Required accesses refers to the number of index (bucket) accesses that are needed in order to reach a predefined recall goal under the *modified collision probability*. Now, we will discuss how to calculate f_Y for all three pruning methods.

In *horizontal pruning*, considering that the elements are uniformly distributed over the sets means that the sets will also be uniformly distributed over the index entries. Then, a random pruning of a ϕ fraction of index entries prunes a ϕ fraction of total index lists. At query time, the queried keys might not collide with any object due to the elimination of that entry from the index. Hence, the collision probability P_1 is modified by the factor $f_h = (1 - \phi)$,

as collision can occur only for non-pruned index entries. For a non-uniform distribution of the sets into the buckets of the index, this pruning factor can be modified using, if known, the cumulative distribution function (CDF) of the elements, i.e., the total posting elements which are removed due to ϕ fraction pruning of index entries.

Similar to *vertical pruning*, the collision probability P_1 is modified by the factor, $f_v = (1 - \phi)$ for horizontal pruning. Although the keys in the index are not pruned, the collision between objects can still be missed due to removal of ϕ fraction of objects from each index list. Considering l_h is the number of keys that we need to look up to reach the predefined recall goal ϱ with modified collision probability, we have $l_h = l_v$. Note that l_h is equal to l_v only when we consider the uniform distribution of data over the index.

In *diagonal pruning*, we use only a ϕ fraction of elements to create the inverted index, i.e., a specific set will be listed under posting lists for only $1-\phi$ of elements from that set. Now, the keys may not collide with a candidate object in this pruning method due to the following two reasons; *(i)* the elements in the key are always chosen from pruned elements of the object, *(ii)* if the elements in the key chosen from pruned elements for few objects but not all. Therefore, we need to find the number of elements removed from the posting lists for both the mentioned reasons. Thus, it is important to know how the keys are generated to build the inverted index. Considering q elements are used to generate the keys, a ϕ fraction of pruning of elements from each set prunes $1 - \binom{k'}{q}/\binom{k}{q}$ (with $k' = (1-\phi)\cdot k$) entries from posting lists, because each object is listed under only $\binom{k'}{q}$ keys. Consequently, the collision probability P_1 is adjusted by the factor $f_d = \frac{k' \cdots (k'-q-1)}{k(k-1)\cdots(k-q-1)}$.

4.3 Cost Model for Query Processing

Query processing is mainly divided into two parts: In the first step, the hash tables are accessed to retrieve candidates (filter step). Then, all candidates are validated to determine the result set $\mathcal{R}$.

The cost of the filtering step depends on the number of index accesses at query time and the length of posting list. Based on the distribution of elements to the individual sets, we can find the expected length of posting lists, referred to as $Post_{len}$. In *probabilistic query processing*, we can find the number of required accesses for different pruning methods, i.e., l_v, l_h, and l_d from Equation 2 by plugging in the specific factors f_Y that were found above.

On the other hand, the required index accesses for *ad-hoc query processing* depends on the number of successful scans, that means, accesses via keys that actually exist. The probability of a key being present in a pruned index is equal to f_Y depending on pruning methods. Considering each index access as a *Bernoulli trial*, we can calculate the number of trials to get the first successful access. Therefore, we calculate the expected number of scans, $E[l_v]$, $E[l_h]$, $E[l_d]$ for ad-hoc query processing which leads to l successful accesses for vertical, horizontal, and diagonal pruning, respectively, by the following equation:

$$E[l_Y] = (1/f_Y) \cdot l \tag{3}$$

Finally, to assess the cost of the validation phase, we need to find the number of candidates retrieved from filtering step which is given by the union of the retrieved elements from l_Y entries of the hash tables. For a total of n objects used to build the index, the expected number of unique candidates is $E_{can}[l_Y] := n(1-(1-Post_{len}/n)^{l_Y})$. Let us consider c_f and c_v are the scanning cost and the cost for the distance calculation for candidate validation respectively. So, the final cost to find $\mathcal{R}$ is given by

$$Cost(l_Y) = (l_Y \cdot Post_{len} \cdot c_f) + (E_{can}[l_Y] \cdot c_v) \tag{4}$$

4.4 Optimizing the Pruning Factor

Now, we discuss how we use the above cost model to determine the optimal value of the pruning parameter ϕ. In Section 1, we have already defined the optimization problem subject to recall guarantee ϱ. Based on different pruning methods, we can estimate the expected number of index accesses (i.e., l_v, l_h, or l_d) needed to ensure the recall requirement, as discussed earlier. As the pruning factor increases, the number of accesses in index also increases. However, there is a bound on number of keys possible to generate from queries. Clearly, if the required l_Y accesses is over this bound, the recall requirement cannot be met. Considering the fixed object sizes k and keys are compounds of q elements, for instance $q = 2$ for pairs, we can generate at most $\binom{k}{q}$ keys from a size-k set (or the query). Hence, the optimization problem is formulated as follows:

$$\begin{aligned} &maximize \quad \phi \\ &\text{subject to} \quad \binom{k}{q} - l_Y = 0, \\ &\qquad Cost(l_Y) \leq \beta \cdot Cost(l) \end{aligned} \tag{5}$$

As the pruning factor is inversely proportional to the number of index accesses, in order to restrict the increasing cost, we fix a parameter β where $\beta \geq 1$ based on application scenario and the user demand. Moreover, $c_f \ll c_v$ due to "curse of dimensionality", thus, the filtering cost can be omitted from the cost model. The validation cost is not inversely proportional to the accesses as it depends on the unique candidates retrieved in the filter step. It is bounded by $O(|\mathcal{R}|)$. Therefore, while omitting the filter cost, the bound of parameter β changes to $\beta > 0$. With this new bound on β the cost constraint looses its significance and by removing this restriction on the access cost from the optimization problem, we can simplify the problem as follows:

$$\phi^* = argmax_\phi \left\{ \binom{k}{q} - l_Y = 0 \right\} \tag{6}$$

5 CASE STUDIES

To demonstrate the versatility of our approach, we now instantiate the above reasoning with specific parameters and probabilities derived from two different scenarios.

Scenario I: Jaccard Distance over Sets. We consider similarity search over sets under the Jaccard distance. We can simply use an inverted index to determine the sets that overlap with a given query set. Let us consider here that we build a pairwise index based on the paired elements from each set. For instance, we have $(4, 5) \rightarrow \langle\tau_1\rangle, \langle\tau_2\rangle$ for the example shown in Table 1. It is known [10] that for this setup (i.e., sets and pairs as keys), the collision probability is $P_1 = \mu/k$ where μ is the minimum overlap for threshold θ in Jaccard distance (same as prefix filtering bound in simple index), i.e., $\mu = 2k\theta/(1+\theta)$. Now, with our reasoning above, we can determine the required number of accesses, l by Equation 1 using P_1 and $m = 2$. Further, we have $q = 2$ to find f_m in the diagonal pruning method and in Equation 6—as two elements are used to create keys.

Scenario II: Kendall's Tau Distance over Rankings. In this scenario we consider rankings instead of sets and the Kendall's tau distance that is basically assessing on how well two rankings agree

Methods	θ for LiveJ			θ for Yago		
	0.1	0.3	0.5	0.1	0.2	0.3
Full scan	52006.1	52006.1	52006.1	37.21	37.21	37.21
Prefix-filtering	23185.1	34045.4	43988.1	19.26	22.78	25.95
Baseline	5105.3	7360.4	9059.5	2.327	2.706	3.067

Table 2: Retrieved candidates for different approaches of query processing in non-pruned index

on the pairwise order of elements. This calls naturally for a pairwise index. But since we now have an order among the elements in a ranking, we can use keys that reflect the order. That means, they keys are pairs as above, but there is a difference between the key (i, j) and the key (j, i): The posting list for the key (i, j) holds those rankings where element i occurs before j, and vice versa for the key (j, i). For instance, entry $(4, 5)$ only holds τ_2 but not τ_1 for the example shown in Table 1. Again, we refer to [10] for obtaining the collision probability P_1 for this scenario. As two elements are used to create the keys for this index, we will set $q = 2$ to find f_d in diagonal pruning methods and in Equation 6.

6 EXPERIMENTS

We have implemented the index structures described above in Java 1.7 and run the experiments using an Intel Core i7@3 GHz machine with 8GB main memory.

The index structures are kept entirely in memory.

We use two datasets, profiles (sets) describing the interests of users, obtained from Life Journal, **LiveJ** [1] for **scenario I**, and **Yago Entity Rankings** for **Scenario II**. In the Yago dataset, we consider 25,000 top-20 rankings that have been mined from the Yago knowledge base, as described in [5]. In the LiveJ dataset, we consider 100,000 profiles of fixed (truncated) size 20. The experimental results are provided by averaging five consecutive experimental runs over 1000 benchmark queries.

In this section, we discuss the experimental studies for both scenarios mentioned in Section 5 to validate our optimization problem. The effect of pruning methods are presented against a baseline approach which is considered as the plain LSH methods on the non-pruned index structures. We also consider the full scan and prefix filtering method in simple index and found that both of them retrieve far more candidates (> 5 times) than the baseline approach, as expected, see the results in Table 2.

In the experimental studies, both pairwise indices are pruned randomly for all three pruning methods. Based on Equation 6, Table 3 represents the theoretically established maximum pruning factor and corresponding number of index accesses, l_Y for predefined recall goal, $\varrho = 0.99$, for both the datasets. As we randomly prune the index, we have $l_v = l_h$, as discussed earlier. Hence, the maximum pruning factor also becomes the same for both methods. We use random pruning as the above modelling also assumes this; other pruning techniqes would require adapting the models, too.

Table 5 and Table 6 discuss the experimental results on query processing for different pruning methods based on theoretically established maximum pruning factors for different θ in similarity search for the case studies. Clearly, we see that the theoretically established ϕ^* fulfills the recall requirement ϱ for probabilistic query processing and thus validates the optimization problem. We also notice that probabilistic query processing retrieves more candidates

[1] http://socialnetworks.mpi-sws.org/data-imc2007.html

Pruning method	LiveJ			Yago		
	θ	ϕ^*	l_Y	θ	ϕ^*	l_Y
Horizontal pruning	0.1	0.8	125	0.1	0.9	53
	0.3	0.8	167	0.2	0.9	68
	0.5	0.7	112	0.3	0.9	90
Vertical pruning	0.1	0.8	125	0.1	0.9	53
	0.3	0.8	167	0.2	0.9	68
	0.5	0.7	112	0.3	0.9	90
Diagonal pruning	0.1	0.5	95	0.1	0.7	73
	0.3	0.5	126	0.2	0.7	97
	0.5	0.4	87	0.3	0.7	130

Table 3: Theoretically established ϕ^*

θ	Yago			LiveJ		
	0.1	0.2	0.3	0.1	0.3	0.5
l	3	5	7	2	4	8
$E[l_v]$	30	50	70	10	20	26.6
$E[l_h]$	30	50	70	10	20	26.6
$E[l_d]$	42.6	71	99.4	8.6	17.39	23.52

Table 4: $E[l_Y]$ for l successful accesses

than the baseline for the LiveJ dataset due to skewed data distribution over index. The length of posting lists in LiveJ lies in $[1, 19335]$. On the other hand, due to more uniform distribution of the data in the Yago dataset, the difference between retrieved candidates for the optimally pruned index and non-pruned index is negligible, which validates our assumption to discarding the cost factor to simplify the optimization problem. From the experimental results we find that the runtime is proportional to the retrieved candidates that also signifies the simplification in optimization problem.

These tables also report on the number of successful index accesses at query time which is bounded by l_Y and $E[l_Y]$ in probabilistic and ad-hoc query processing respectively. Table 4 presents $E[l_Y]$ to reach l successful accesses in ad-hoc query processing based on Equation 3. Table 4 also mentions the required access l for the baseline approach based on Equation 1.

In Table 5 and Table 6, experimental results show that the ad-hoc query processing always retrieves less candidates than probabilistic query processing and does not assure the required recall requirement due to its ad-hoc characteristic. However, this method reaches the recall requirement only when #total accesses in ad-hoc query becomes almost same as the expected number of index accesses given in Table 4. It also validates/supports our probabilistic analysis for ad-hoc query processing. For example, in vertical pruning for both scenarios, ad-hoc query processing reaches recall requirement ϱ and #total accesses $\approx E[l_v]$ (highlighted in blue). As horizontal pruning does not discard any keys from the index, the #total accesses in ad-hoc query processing is the same as l in baseline method which is far less than $E[l_h]$ and, thus, does not reach the recall requirement.

7 RELATED WORK

Many authors have proposed locality sensitive hash families for different metric and Euclidian distance functions [2, 6]. In [14], a survey over LSH methods for similarity search under many different distance functions is presented. Query-aware LSH methods are discussed in [4, 7, 10], where hash functions are selected at query time. Such approaches commonlystore more hash tables than the tables that are actually used at query time, which leads to an inefficient space usage. In our prior work [10], we presented a query-driven LSH method and proposed two hash families for Kendall's

Pruning method	θ	Probabilistic query processing time	#Candidates	recall	# successful accesses	Ad-hoc query processing time	#Candidates	recall	# Total accesses	# successful accesses	# baseline candidates in
Horizontal	0.1	11.17	10031.3	100	24.6	3.4	3806.7	100	9.45	2	5105.3
	0.3	11.54	13257.0	100	33.93	4.4	5163.3	100	19.39	4	7360.4
	0.5	13.39	14452.2	100	33.59	7.4	7822.5	99.9	26.29	8	9059.5
Vertical	0.1	14.0	11252.9	100	125	1.03	1142.2	51.3	2	2	5105.3
	0.3	9.8	12208.7	100	167	1.67	1926.9	64.3	4	4	7360.4
	0.5	11.0	14001.9	100	112	4.08	3998.9	93.6	8	8	9059.5
Diagonal	0.1	10.38	10378.3	99.5	79.69	1.24	1309.1	37.3	2.63	2	5105.3
	0.3	11.06	11512.7	100	104.58	1.99	2098.4	47.3	4.94	4	7360.4
	0.5	11.32	13003.1	99.7	76.84	3.52	3938.9	61.0	9.61	8	9059.5

Table 5: Query processing in pruned pairwise index on LiveJ based with ϕ^* for $\varrho = 99\%, k = 20$

Pruning method	θ	Probabilistic query processing time	#Candidates	recall	# successful accesses	Ad-hoc query processing time	#Candidates	recall	# Total accesses	# successful accesses	# baseline candidates
Horizontal	0.1	0.027	2.37	100	5.2	0.017	2.05	99.8	30.33	3	2.327
	0.2	0.050	2.39	99.8	6.8	0.031	2.39	99.8	50.63	6.7	2.706
	0.3	0.030	2.65	99.8	8.9	0.040	2.65	99.7	70.82	6.9	3.068
Vertical	0.1	0.040	3.63	100	53	.008	1.52	93.51	3	3	2.327
	0.2	0.050	4.00	100	68	0.013	1.77	95.02	5	5	2.706
	0.3	0.192	4.46	100	90	0.016	1.96	94.1	7	7	3.068
Diagonal	0.1	0.037	2.32	99.1	8.09	0.018	1.56	90.8	30.28	3	2.327
	0.2	0.047	2.61	99.9	10.6	0.035	1.87	94.5	498.7	5	2.706
	0.3	0.055	2.84	99.5	14.3	0.048	2.10	94.6	67.61	7	3.068

Table 6: Query processing in pruned Unsorted pairwise index based on Yago with ϕ^* for $\varrho = 0.99$, $k = 20$

tau distance, by projecting the ranking objects over the elements it contain. Therefore, many hash tables are needed to store but only few are used at query time which also leads to rather inefficient space management. Different variants of LSH methods address this problem [8, 9]. Although the ad-hoc query processing scheme seems to be similar like query-processing in the multi-probe LSH approach [9], they are very different in nature. In multi-probe LSH, a fixed sequence of perturbed vectors are used to find hash buckets near to the buckets where query is mapped, whereas ad-hoc query processing looks up the hash entries in the bucket of different hash functions until the key is found in the pruned index.

Different static index pruning methods are also developing in parallel for searching top-k documents in for creating efficient search engine. The work by Soffer et al. [12] contributes a term-based pruning method based on tf-idf score, Büttcher and Clarke [1] discuss document-centric index pruning. As mentioned earlier, be believe such attempts are fully orthogonal, but leave any integration for future work, as it goes beyond the scope of this paper.

8 CONCLUSION AND OUTLOOK

In this paper, we discussed three different ways to prune inverted indices and presented a probabilistic analysis of the effect of index pruning to query processing, by exploiting the relatedness of inverted indices to locality sensitive hashing (LSH). Based on this analysis, we were able to formalize an optimization problem that determines the optimal pruning factor, considering a user-defined recall requirement and cost of query processing. Experimental evaluation was conducted over two case studies that validates our optimization problem. The evaluation results showed that the optimal pruning factor is quite high ($\geq$ 80%) for small distance threshold in similarity search ($\theta \leq 0.3$), and ad-hoc query processing in horizontal pruning outperforms for optimal pruning factor in both scenarios. As future work we would like to investigate the potential of combining the approach with index compression techniques and more application-driven, non-randomized pruning. Further, we would like to extend this approach to work with rankings and sets that can vary in size.

REFERENCES

[1] Stefan Büttcher and Charles L. A. Clarke. 2006. A document-centric approach to static index pruning in text retrieval systems. In *CIKM*.
[2] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*.
[3] Sven Helmer and Guido Moerkotte. 2003. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.* 12, 3 (2003).
[4] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.
[5] Evica Ilieva, Sebastian Michel, and Aleksandar Stupar. 2013. The essence of knowledge (bases) through entity rankings. In *CIKM*.
[6] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*.
[7] Herve Jegou, Laurent Amsaleg, Cordelia Schmid, and Patrick Gros. 2008. Query adaptative locality sensitive hashing. In *IEEE ICASSP*.
[8] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. 2014. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. *PVLDB* 7, 9 (2014), 745–756.
[9] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *VLDB'07, University of Vienna, Austria, September 23-27*.
[10] Koninika Pal and Sebastian Michel. 2016. Efficient Similarity Search across Top-k Lists under the Kendall's Tau Distance. In *SSDBM*.
[11] Gleb Skobeltsyn, Flavio Junqueira, Vassilis Plachouras, and Ricardo A. Baeza-Yates. 2008. ResIn: a combination of results caching and index pruning for high-performance web search engines. In *SIGIR*.
[12] Aya Soffer, David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, and Yoëlle S. Maarek. 2001. Static Index Pruning for Information Retrieval Systems. In *SIGIR*.
[13] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*.
[14] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. *CoRR* abs/1408.2927 (2014).
[15] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006).