LSH Forest: Self-Tuning Indexes for Similarity Search

Mayank Bawa
Stanford University
Stanford, CA 94305
bawa@db.stanford.edu

Tyson Condie
U. C. Berkeley
Berkeley, CA 94720
tcondie@eecs.berkeley.edu

Prasanna Ganesan
Stanford University
Stanford, CA 94305
prasanna@db.stanford.edu

ABSTRACT

We consider the problem of indexing high-dimensional data for answering (approximate) similarity-search queries. Similarity indexes prove to be important in a wide variety of settings: Web search engines desire fast, parallel, main-memory-based indexes for similarity search on text data; database systems desire disk-based similarity indexes for high-dimensional data, including text and images; peer-to-peer systems desire distributed similarity indexes with low communication cost. We propose an indexing scheme called LSH Forest which is applicable in all the above contexts. Our index uses the well-known technique of locality-sensitive hashing (LSH), but improves upon previous designs by (a) eliminating the different data-dependent parameters for which LSH must be constantly hand-tuned, and (b) improving on LSH's performance guarantees for skewed data distributions while retaining the same storage and query overhead. We show how to construct this index in main memory, on disk, in parallel systems, and in peer-to-peer systems. We evaluate the design with experiments on multiple text corpora and demonstrate both the self-tuning nature and the superior performance of LSH Forest.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: search process; H.3.4 [**Systems and Software**]: Distributed systems, information networks, Performance evaluation (efficiency and effectiveness)

General Terms

Algorithms, Performance, Similarity, Search

Keywords

Similarity indexes, peer-to-peer (P2P)

1. INTRODUCTION

Performing similarity search to find objects most similar to a given object is a classical problem with many practical applications. For example, a Web search engine wants to find pages most similar to a given Web page. A database system wants to support similarity queries over text, image or video data. A peer-to-peer content-sharing system wants to support queries that find content, or peers themselves, that are most similar to a given content instance. Other application domains [20] include data compression, data mining, machine learning, pattern recognition and data analysis.

Given a small dataset size, one could simply compare each object to the query in order to find the most similar objects. However, such an approach becomes infeasible due to linear querying costs even for medium-sized datasets. Furthermore, if the cost of computing similarity between pairs of objects is high, then even a partial scan of the dataset can overwhelm the processing resources available.

The solution is to develop indexes that, given any query, will select only a small set of "candidate" objects to compare the query against. The "best" indexes have the following properties:

A *Accuracy:* The set of candidates retrieved by the index should contain the most similar objects to the query.

B *Efficient Queries:* The number of candidates retrieved must be as small as possible, to reduce I/O and computation costs.

C *Efficient Maintenance:* The index should be built in a single scan of the dataset, and subsequent inserts and deletes of objects should be efficient.

D *Domain Independence:* The index should require no effort on the part of an administrator to get it working on any data domain; there should be no special tuning of parameters required for each specific dataset.

E *Minimum Storage:* The index should use as little storage as possible, ideally linear in the data size.

The B+ Tree is a classic example of one of the "best" indexes for range queries on one-dimensional ordered domains. The B+ tree is always accurate, returning exactly the query results. Queries are efficient, requiring at most $O(\log n)$ disk reads, and one sequential scan, in an n-object dataset. It supports efficient inserts and deletes of data with $O(\log n)$ disk writes per insert/delete, while keeping the tree balanced at all times. It is completely domain-independent, requiring only a specification of the comparison function in the domain. Finally, it uses only $O(n)$ storage space, thus providing all five properties listed above.

In this paper, we design an index for approximate *similarity-search* queries that meets the desiderata listed above. Our index is based on the influential locality-sensitive hashing (LSH) scheme of Indyk and Motwani [23]. The basic idea behind LSH is the following: objects are hashed using special locality-sensitive hash functions, such that "similar" objects are much more likely to collide (hash to the same bucket) than dissimilar objects. Many similarity measures have corresponding LSH functions that provide this property. Examples of such similarity measures include the Jaccard coefficient, the Hamming metric and the l_1 and l_2 norms [23].

At query time, the query q is also hashed to a bucket; objects which hash to the same bucket are retrieved as "candidate" answers, their actual similarity to the query is computed, and the most similar among them are returned as answers to the query. The results returned by LSH are guaranteed to have a similarity within a small error factor ϵ of the optimal, for any fixed $\epsilon > 0$.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

The basic LSH scheme as described above satisfies Properties [A], [B] and, to some extent, [C]. However, it fails in Property [D] as an administrator must tune various index parameters (size of signatures k, distance to nearest neighbors r, and number of indexes l), which are a function of the data domain, for the accuracy guarantees to hold [23, 20]. These parameters also depend on the number of objects being stored, and thus may have to be changed if the corpus size changes over time (violating Property [C]). In addition, there is a trade-off between the storage space required (Property [E]) and the accuracy that can be guaranteed for queries (Property [A]). Guaranteeing accuracy for all queries requires a large number of indexes, each tuned with a different set of parameters, making the storage requirement proportional to $\frac{1}{\epsilon}$ [20] which violates Property [E]; on the other hand, with limited storage space, one cannot guarantee good accuracy for all queries which can be a problem especially in domains with skewed data distributions.

Summary of Results: We present a new indexing scheme called the LSH Forest which uses locality-sensitive hashing while avoiding the problems listed above. The LSH Forest improves the theoretical guarantees on query performance, ensuring accuracy for all queries without a corresponding blow-up in storage space (thus simultaneously offering Properties [A] and [E]). It also eliminates the need for tuning the index in a domain-dependent fashion, or whenever the corpus size changes (thus offering Properties [C] and [D]). All these properties are achieved while retaining the efficient querying offered by the basic LSH scheme (Property [B]).

The LSH Forest can be implemented in a variety of settings. In this paper, we describe how it may be constructed as a main-memory index, as a disk-based index, as a parallel index, and as a distributed index for P2P systems. One particularly interesting application we consider is in P2P systems where the objects being indexed are the peers themselves, and queries attempt to find the most similar peers to a given object/peer; we describe an efficient implementation for this case where the *overlay network* interconnecting peers is structured to reflect similarity information, avoiding the need for an explicit similarity index entirely.

We evaluate the LSH Forest in the context of similarity search over text documents, using the standard test collections from TREC and Reuters. We demonstrate that the LSH Forest offers a clear improvement over even the best-tuned standard LSH index in terms of answer quality, while retaining the same storage and query costs and avoiding the need for all parameter-tuning.

Organization: We discuss related work in Section 2. Section 3 defines the basic problem, and Section 4 presents an overview of the basic LSH scheme for similarity search. We present our LSH Forest indexing scheme in Section 5 and discuss how it may be implemented in different contexts. Section 6 describes the experimental methodology evaluating the use of the LSH Forest for the problem of text similarity search. We present our experimental evaluation of the index in Section 7.

2. RELATED WORK

Similarity search has been a topic of much research in recent years. The work in literature can be broadly classified into four categories based on the following notions of similarity:

- Nearest-neighbor queries,
- Duplicate detection,
- Link-based similarity search, and
- Defining object representation

Nearest Neighbors: Much work in the database literature has considered tree-based indexing methods for efficiently indexing tuples represented as points in a metric space. The indexes have proven to be useful for low and moderate dimensional spaces. A common strategy for indexing high dimension points is to map the data points to lower dimensional space and perform similarity searches in that space [15]. Some strategy is then used to group data together using a bounding object. The bounding objects are organized in a tree structure to support efficient querying. The type of bounding object and the method of constructing and maintaining the trees vary widely: minimum bounding rectangles (e.g., K-D-B-Trees [30], R-Trees [21], R*-Trees [24], X-trees [4]), polyhedra (e.g., Pyramid-Trees [3]), and hyperspheres (e.g., SS-Trees [33]). Any fixed bounding object however never fits real data tightly, causing data to be sparse, and not necessarily uniformly distributed, within bounding objects. In high dimensions, the query point overlaps with a lot of bounding objects, many of which contain no relevant data points, resulting in non-trivial paging costs at query time. In particular, Weber et. al. [32] show, both empirically and theoretically, that *all* current indexing techniques based on space partitioning degrade to linear search for sufficiently high dimensions. This situation poses a serious obstacle to the development of Web-scale content similarity search systems based on spatial indexing.

Duplicate Detection: Several algorithms (e.g., [5, 6, 8, 10, 16, 31]) have been studied to find *nearly identical* documents in a large corpus. In all algorithms, there is a user-provided threshold parameter, that bounds the “dissimilarity” beyond which documents are of no interest, which is empirically adjusted to suit the target domain. The algorithms rely heavily on the threshold parameter being *very low* to aggressively prune possible candidate pairs of similar documents. If the threshold parameter is higher (as for general similarity search), processing is overwhelmed with large numbers of (false) candidates, making the algorithms unscalable for our domain.

Link-Based Similarity Search: Many commercial search engines provide a “find related pages” feature for pages on the Web, but the details of their algorithms are not publicly available. Dean and Henzinger [14] present two algorithms (Companion and Cocitation) that rely only on the hyperlink structure of the Web to identify *related* Web pages. Companion extends Kleinberg’s HITS algorithm [25] while Cocitation finds pages that frequently co-occur with the query page, to find related pages. The algorithms suffer from the drawback that pages with few in-links will not have sufficient co-citations to either be allowed in queries, or be returned as results to queries. Furthermore, the vicinity graph of the query page has to be constructed (to compute hubs and authority scores in Companion, and to count co-occurrences in Cocitation) resulting in non-trivial processing costs at query time.

Defining Object Representation: Often, deducing a compact representation of an object is an interesting problem by itself. Again in the Web context, the ideas of “topical locality” provided by co-occurring hyperlinks have been used and studied in several works [9, 13, 19]. Haveliwala et al. [22] show that using words appearing inside or near an anchor of a hyperlink to represent a linked Web page is most useful for *similarity* search on the Web. In [22], a Web crawl is pre-processed to construct an LSH-based index [7, 11] that maps each page p to a set of pages S that are similar to p above a certain threshold parameter. At query time, the index is used to retrieve S and return the most similar documents. Our focus in this work is on replacing the simple LSH-based index by a better structure that avoids all the drawbacks of LSH discussed earlier.

3. PROBLEM SETUP

Our objective is to design an index to perform similarity search on a dynamic set of objects. As is conventional, we refer to objects as *points* in some high-dimensional space, and the distance between a pair of points is defined by some distance function $D(.,.)$. (Similarity between points is inversely related to their distance.)

A similarity search query q that desires m results is called an $m-$nearest neighbor query, and is required to return the m points in the data set that are closest to q according to the distance function D. Finding the exact solution to an m-nearest-neighbor query is known to be hard [23]. We relax our requirements to find only *approximate* nearest neighbors: for a pre-specified value $\epsilon > 0$, we wish to find m neighbors, such that the distance from q to the i^{th} nearest neighbor returned is at most $(1+\epsilon)$ times the distance from q to its true i^{th} nearest neighbor.

Our problem is to design an index structure that enables efficient ϵ-approximate nearest-neighbor queries, efficient building of the index, efficient insertion and deletion of points, and complete domain independence, all while ensuring minimal use of storage. We note that our solution (and, for that matter, any solution) will not work for an *arbitrary* choice of the distance function D; however, it does work for any choice of D for which there is a corresponding LSH family, as we discuss later on.

4. LSH: OVERVIEW AND PROBLEMS

We first present an overview of locality-sensitive hashing (LSH), and how it is used in constructing indexes for similarity search. We will then describe the difficulties that emerge in using such indexes, in order to motivate our solution that we present in the next section. Our treatment of LSH is drawn from, and follows the notation of [23, 20] with simplifications where appropriate; we refer the reader to these papers for more details.

The intuition behind LSH-based indexes is to hash points into buckets, such that "nearby" points are much more likely to hash to the same bucket than points that are far apart. We could then find the approximate nearest neighbors of any point, simply by finding the bucket that it hashes to, and returning the other points in the bucket. In fact, Broder et al. [8] use this intuition to find *nearly identical* documents in Web crawls. Translating this intuition to find *similar* documents requires a concrete definition of which points are "nearby", and special, locality-sensitive hash functions that ensure that only nearby points collide.

4.1 Locality-Sensitive Hash Families

We start by defining the requirements for a hash function family to be considered locality-sensitive.

DEFINITION 1. *[23, 20] A family $\mathcal{H}$ of functions from a domain $\mathcal{S}$ to a range U is called (r, ϵ, p_1, p_2)-sensitive, with $r, \epsilon > 0$, $p_1 > p_2 > 0$, if for any $p, q \in \mathcal{S}$, the following conditions hold:*

- *if $D(p, q) \leq r$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \geq p_1$,*
- *if $D(p, q) > r(1+\epsilon)$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \leq p_2$.*

Intuitively, the definition states that "nearby" points within a distance r of each other are likely to collide (with probability p_1), while "distant" points more than $r(1+\epsilon)$ apart have only a smaller probability (p_2) of colliding. (The probabilities are computed over the random choice of hash function h from $\mathcal{H}$.) It is possible to define such locality-sensitive hash families for many different distance functions D, including the Jaccard measure, the Hamming norm, and the l_1 and l_2 norms [23]. We will discuss the LSH family for the Jaccard measure in greater detail in Section 6.

4.2 The LSH Index

Assume that there is some family of LSH functions $\mathcal{H}$ available. We can then use $\mathcal{H}$ to construct a set of hash tables as below, that we collectively refer to as an *LSH index*.

1. Choose k functions $h_1, h_2, \ldots h_k$ uniformly at random (with replacement) from $\mathcal{H}$. For any $p \in S$, place p in the bucket with label $g(p) = (h_1(p), h_2(p), \ldots h_k(p))$. Observe that if each h_i outputs one "digit", each bucket has a k-digit label.
2. Independently perform step (1) l times to construct l separate hash tables, with hash functions $g_1, g_2, \ldots g_l$.

Step (1) concatenates k different hash functions together to identify the hash bucket; if two "distant" points had a probability p_2 of collision with one hash function, their collision probability drops to just p_2^k with the concatenation, which becomes negligibly small for large k. It is easy to see that a small value of k increases the number of false positives created by dissimilar points colliding. However, a larger value of k has the side-effect of lowering the chances of even nearby points colliding. Therefore, to ensure enough "good" collisions occur, step (2) constructs *multiple* hash tables.

Index Maintenance and Usage: Maintaining the index as points are inserted and deleted is straightforward. When a query q is issued desiring m nearest neighbors, it is hashed to find collisions in each of the hash tables. A fixed number cl of colliding points (for some pre-determined choice of constant c) are examined, and the m nearest points among them are returned as the answers to q.

Index Properties: Assume that the query q requires only one nearest neighbor, and that we somehow knew the distance r from q to its true nearest neighbor. If $\mathcal{H}$ is $(r, \epsilon, p_1, p_2)-$sensitive, it can be shown that, for suitable choices of k and l as a function of r, ϵ and the number of points n, the index returns an ϵ-approximate nearest neighbor for q. Furthermore, the number of points examined by the query is strictly sub-linear in n and the storage overhead for the index is sub-quadratic in n.

Observe the important caveat however: *we need to know the distance r from the query to the nearest neighbor in order to select k and l.* In other words, once we have frozen a definition of what constitutes a "nearby" point (distance less than r) and what constitutes a far-away point (distance greater than $r(1+\epsilon)$), it is possible to construct a tuned index that returns approximate nearest neighbors. However, if different queries have their nearest neighbors at different distances, the LSH index cannot provide good performance for all of them.

A theoretical work-around for this problem is to construct *many* LSH indexes, each tuned for a different value of r, say $r = r_0, r_0(1+\epsilon), r_0(1+\epsilon)^2, \ldots$, thus ensuring that there is one index that works well for every query. Unfortunately, this results in a large blow-up in the storage cost. (An alternative theoretical solution is to construct a structure known as ring-cover trees [23] but it does not appear feasible in practice.)

In practice, it is recommended [20] to just choose a "good" value of r that captures the nearest-neighbor distance for most queries, and construct the LSH index with this value of r. When nearest-neighbor distances are relatively uniform, this turns out to be sufficient to obtain good results. We also note that, while the theoretical arguments are for 1-nearest neighbor queries, the indexes work well for m-nearest neighbors as well (with the number of retrieved candidates changing appropriately).

4.3 Difficulties with the LSH Index

We now summarize the key theoretical and practical difficulties with the use of the LSH index. On the theoretical side, the LSH index suffers from the following problems:

A Each index has two parameters k and l which depend on the number of points n, as well as r and ϵ. Therefore, the index may need to be reconstructed with different parameters whenever n changes by a sufficiently large amount, or when the data characteristics change.

B Obtaining ϵ-approximate nearest neighbors for all queries requires the construction of a large number of indexes, which can be expensive, in both storage and processing costs.

On the practical side, the first problem could be tackled simply by setting k and l to be constants, and hoping that variations in the different parameters does not affect performance too badly. It turns out that the performance does not depend crucially on l, so long as l is chosen to be large enough, say 10 [20].

The value of k, on the other hand, is critically dependent on the data characteristics; in some cases (like the ones we present in Section 7), it is also strongly affected by n. Thus, the LSH index needs to be (a) tuned carefully, using data analysis to find the right value of k, which is a non-trivial task; and (b) re-tuned periodically to sustain performance with changing data size and characteristics.

The second problem – avoiding constructing a large number of indexes – can be combated by building just one index for a good choice of r. However, the price we must pay is the need to set yet another tuning parameter r that can critically affect the index performance. Moreover, the index will not work well for all queries if data is skewed with high variance in nearest-neighbor distances.

5. THE LSH FOREST

Having seen that the LSH index suffers from the need for tuning, continuous re-tuning, as well as the inability to provide strong quality guarantees for *all* queries, we describe how these problems may be overcome by a different LSH-based approach to constructing indexes that we call the LSH Forest. We will first present the logical data structure to be constructed, explaining how queries are executed, and the structure maintained. Later, we discuss how the logical structure may be realized in different implementation scenarios.

5.1 A Logical View

Recall that the LSH index placed each point p into a bucket with label $g(p) = (h_1(p), h_2(p), \ldots, h_k(p))$, where $h_1, h_2, \ldots, h_k$ are chosen randomly with replacement from $\mathcal{H}$. We could think of $g(p)$ as being a k-digit *label* assigned to point p. Now, instead of assigning fixed-length labels to points, we let labels be of *variable length*; specifically, each point's label is made long enough to ensure that every point has a distinct label.

It is possible that some points are so similar that assigning distinct labels to them would make the labels very long. To avoid this problem, we impose a maximum label length ${k_m}$[1]. The variable-length label of a point is generated in the obvious fashion: We let $h_1, h_2, \ldots, h_{k_m}$ be a sequence of k_m hash functions drawn independently and uniformly at random from $\mathcal{H}$. The length-x label of a point p is given by $g(p, x) = (h_1(p), h_2(p), \ldots, h_x(p))$.

The Data Structure: We can then construct a (logical) prefix tree on the set of all labels, with each leaf corresponding to a point. We call this tree the *LSH Tree*. Our indexing structure simply consists of l such LSH Trees, each constructed with an independently drawn random sequence of hash functions from $\mathcal{H}$. We call this collection of l trees the *LSH Forest*.

EXAMPLE 5.1. *Figure 1 shows an LSH Tree that contains four points, with each hash function producing one bit as output. The leafs of the tree correspond to the four points, with their labels marked inside. The shaded circles correspond to internal nodes. Observe that the label of each leaf simply represents the path to it from the root. Also observe that not all internal nodes need to have two children; some internal nodes may have only one child (for example, the right child of the root). In general, there is no limit on the number of internal nodes in a prefix tree with n leaves, since we can have long chains of internal nodes. Later on, we will discuss how to represent such prefix trees compactly.*

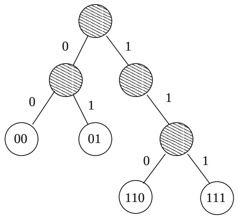

Figure 1: A prefix tree on the set of LSH labels

```
ALGORITHM DESCEND($q, x_i, s$)
    ▷ Args: query $q$ at level $x_i$ on node $s$
    if ($s$ is leaf) {
        Return $(s, x_i)$
    } else {
        $y = x_i + 1$
        Evaluate $g_i(q, y)$
        $t$ = Child node from branch labeled $h_y(q)$
        $(p, z)$ = DESCEND($q, y, t$)
        Return $(p, z)$
    }
```

Figure 2: The top-down phase in query processing on LSH Tree T_i initiated with DESCEND**($q, 0, root_i$).**

Queries: Consider an LSH Forest consisting of l prefix trees built on a set of points. A query for the m nearest neighbors of a point q is answered by traversing the LSH Trees in two phases. In the first top-down phase, we descend each LSH Tree T_i to find the leaf having the largest prefix match with q's label as shown in Figure 2.

Let $x := max_i^l\{x_i\}$ be the maximum (bottom-most) level of leaf nodes across all l trees. In the second bottom-up phase, we collect M points from the LSH Forest, moving up from level x towards the root synchronously across all LSH Trees as shown in Figure 3. For 1-nearest neighbor queries, we simply set $M = cl$ for a small constant c (we count duplicates as part of M). For m-nearest neighbor queries, we have more choices; for example, M can be set dynamically at runtime to ensure that there are at least m distinct points returned and $M \geq cl$. The M points are then ranked in order of decreasing similarity with q from which the top m distinct points are returned as an answer to the query q.

Inserts: Point insertions are executed independently on each LSH Tree. A significant part of the process of inserting a point p is the top-down search just described. When a leaf node with point p' is reached, the labels of both p and p' are extended with additional digits until they become distinct and occupy unique leaf nodes.

Deletes: Deletions are the converse operation of inserts. The top-down phase of the search algorithm is executed to reach the appropriate leaf node, which can then be removed from the tree. A bottom-up traversal of the tree towards the root potentially contracts the labels of internal and leaf nodes.

[1] This problem arises in the LSH index too, where the corresponding assumption is that all points are a minimum distance r_0 apart. In both cases, the core difficulty is that it is impossible to distinguish between two points that are arbitrarily close to each other by means of hashing.

```
ALGORITHM SYNCHASCEND($x[1,\ldots,l], s[1,\ldots,l]$)
    ▷ Args: $x_i$ values and corresponding leaf nodes $s_i$ for each $T_i$
    $x = max_i^l\{x[i]\}$
    $P = \phi$
    while ($x > 0$ and ($|P| < cl$ or $|distinct(P)| < m$)) {
        for ($i = 1 ; i \le l ; i++$)
            if ($x[i] == x$) {
                $P = P \cup Descendants(s[i])$
                $s[i] = Parent(s[i])$
                $x[i] = x[i] - 1$
            }
        $x = x - 1$
    }
    Return $P$
```

Figure 3: The synchronous bottom-up phase in query processing initiated with arguments returned by DESCEND**.**

Theoretical Guarantees: For a proper choice of $\mathcal{H}$ and l and with $c = 3$, we can show that queries using the LSH Forest return ϵ-approximate neighbors, irrespective of the distance of the query from its nearest neighbor. We start by defining the special class of LSH functions $\mathcal{H}$ that we require for the LSH Forest.

DEFINITION 2. *A family $\mathcal{H}$ of functions from $\mathcal{S}$ to U is called $(\epsilon, f(\epsilon))$-sensitive in the range (a, b), if for any $p, q \in \mathcal{S}$, with $a < D(p, q) < b$, and any $r : a < r < b$, there exist $p_1 > p_2 \geq 0$ such that the following conditions hold:*

- *if $D(p, q) \leq r$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \geq p_1$,*
- *if $D(p, q) > r(1 + \epsilon)$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \leq p_2$*
- $\log(1/p_1) > f(\epsilon) \log(1/p_2)$

Note that the above definition is almost identical to the earlier definition of LSH families, except that the parameter r has been eliminated. We also note that all the "typical" LSH families used in the basic LSH scheme (and that we describe later) satisfy the more restrictive definition above. Then, with such a choice of $\mathcal{H}$ the theorem below shows that LSH Forest returns ϵ-approximate nearest neighbors for a suitable choice of l (with say, $c = 3$.)

THEOREM 5.1. *With an $(\epsilon, f(\epsilon))$-sensitive hash family $\mathcal{H}$ in the range (a, b), setting $l = n^{f(\epsilon)}$ ensures that any nearest-neighbor query q returns ϵ-approximate neighbors using the LSH Forest with a non-zero probability greater than a constant C, so long as the distance from q to its nearest neighbor is in the range (a, b).*[2] □

The above theorem is almost exactly identical to the corresponding theorem for the LSH Index [23, 20], with the only difference being that the parameter k has been eliminated, and the guarantee holds for *all* queries. The restriction on the distance from q to its nearest neighbor is a technical condition arising from the fact that, with limited storage space, it is hard to distinguish between nearby and far-away points when r is too small or too large. The same technical conditions manifest themselves in the original LSH index structure as well, and do not affect us in practice.

Practical Implications: We note that the guarantees provided by the theorem are fairly weak; the number of points examined by the query is only guaranteed to be sub-linear in n, the storage cost is sub-quadratic (as we will show in the next subsection), and l is also a sub-linear function of n. All these guarantees are identical to that for the LSH index [20]. In reality, however, the picture is much brighter. A small value of l, say $l = 10$, proves sufficient to obtain excellent performance, requiring examination of only a constant number of points per query and only a small linear storage overhead. One of the reasons for this practical improvement is that the theorems are proved for arbitrary sets of points; real data sets exhibit much more structure which makes them easier to index.

5.2 Main-Memory Implementation

The LSH Tree as we have described it is not efficient in its usage of storage space, since the number of internal nodes could potentially be very large. For example, if there were just two points which shared a common 100-bit prefix, 102 internal nodes would be created to support just the two points. However, this problem is easily fixed: a long chain of "logical" internal nodes is compressed into just one node that stores the path to it from the previous node. For example, in Figure 1, we can compress the right child and grandchild of the root into a single node, without affecting any of the LSH Tree operations. In fact, this compression trick is well-known and prefix trees constructed in this fashion are called PATRICIA tries [29].

With this modification, the number of internal nodes become exactly one less than the number of leaf nodes, thus ensuring that the storage cost of a single tree is linear in the number of points n.

Note that even the resulting prefix tree may not be balanced, and queries can therefore require many pointer traversals to navigate to the bottom of the tree. We note two things however: (a) The number of downward pointer traversals is never too large, being bounded by k_m. (In some of our experiments, k_m was as small as 15.) (b) The primary bottleneck in main-memory similarity search is the *computation* of similarity between the candidates and the query. The query cost is therefore dominated by the number of candidates returned and not by pointer traversals.

5.3 An Efficient Disk-based Implementation

Let us now consider how to implement the LSH Forest as a disk-based index for large data sets. The core problem in developing an efficient disk-based index is to lay out the prefix tree on disk in such a fashion as to minimize the number of disk accesses required to navigate down the tree for a query, and also to minimize the number of *random* disk seeks required for all index operations.

There are a slew of disk-based data structures for storing and querying string data (e.g., Prefix B-Trees [2], paginated B-Trees [28], String B-Trees [17], Index Fabric [12]), which can all be used in our context. Of these, Prefix B-Trees are the simplest and enjoy the most usage, and we describe them here.

A Prefix B-Tree is essentially a B-tree that stores strings sorted in lexicographic order. However, the storage cost of the B-tree is reduced by using two compression schemes at internal nodes [27]: *head compression* that factors out a common prefix from all index entries, and *tail compression* that selects a short index term to act as separators for keys of one data node from those of its siblings. The compression produces significant improvements in performance; the size of the index is smaller overall, branching factor at each internal node is increased, and hence fewer disk I/O's are required to reach data at the leaf nodes. Ferragina et al. [17] mention that Prefix B-Tree performance is good for *bounded-length keys* (≤ 255 bytes) [3]: a single key search or update takes $O(\log_B n)$ disk access-

[2] Repeating the construction $O(1/\delta)$ times leads to a high probability of success greater than $1 - \delta$ for any $\delta > 0$ in theory. However, this means building more copies of the index, which turn out to be unnecessary in practice [20].

[3] This bound accommodates labels with $k = 2040$ and will be sufficiently large for most domains we foresee.

```
ALGORITHM ASYNCHASCEND($x_i, s_i$)
    ▷ Args: $x_i$ value and corresponding leaf node $s_i$ for $T_i$
    $P = \phi$
    while ($x_i > 0$ and $|P| < max(c, m)$) {
        $P = P \cup Descendants(s_i)$
        $s_i = Parent(s_i)$
        $x_i = x_i - 1$
    }
    Return $P$
```

Figure 4: The asynchronous bottom-up phase in query processing on LSH Tree T_i initiated with arguments returned by DESCEND **on T_i.**

es where B is the branching factor of an internal node and n is the number of points in the dataset.

We now show how the logical operations can be efficiently implemented on a Prefix B-Tree, augmented appropriately with additional information at each interior node.

Insertions: The insertion of a point p is just the regular Prefix B-Tree insert of p's label with additional operations at the leaf node. The labels of p and other points at the leaf node are extended until they become distinct. This requires changes of keys recorded at the leaf node only; the internal nodes are unaffected as the label prefixes remain unchanged. The cost of an insert is thus $O(l \log_B n)$ disk accesses followed by at most lB label extensions.

Deletions: The deletion of a point p is just the regular Prefix B-Tree delete of p's label with no additional operations at the leaf node. Specifically, we do *not* shrink the labels of other nodes remaining in the tree. The cost of a delete is thus $O(l \log_B n)$ disk accesses.

Queries: A query asking for m nearest neighbors to a point q is executed in two phases. In the first phase, we descend the trees searching for a Prefix B-Tree key with label digits $g_i(q, x_i)$. The search terminates at a leaf node where we locate the point s_i with the largest prefix overlap (of $x_i - 1$ digits) with q's label. In the second phase, some M points are to be collected across all trees synchronously. Let $x = max_i^l\{x_i\}$ be the maximum overlapping prefix length. Most of the M points will share a prefix with s_i and hence are expected to lie within the leaf node of the LSH Trees, or its siblings. If the leaf node is exhausted, we step back to the parent node to fetch and explore siblings of the leaf. The process is repeated until M points are collected with m distinct points. The M points are ranked by similarity scores, and the top m are returned as answer. The cost of a query is the sum of its first phase (a total of $O(l \log_B n)$ disk accesses and $O(l \log B)$ comparisons in a binary search to locate s_i) and its second phase (a total of $O(M/B)$ disk accesses). Observe that the number of block accesses for the second phase is optimal within constant factors.

Asynchronous Queries: Observe that the query operation discussed above performs a synchronous exploration of the l LSH Trees. The nearest neighbors are retrieved in lock-step from each successive level in the different trees, causing random disk I/O requests. We now present a variant that breaks synchrony by relaxing the requirements imposed by Theorem 5.1. Instead of selecting M=cl points from across l LSH Trees, the variant picks $max(c, m)$ points independently from each LSH Tree as shown in Figure 4. Although the variant does not guarantee ϵ-approximate neighbors, we show in Section 7 that it works almost as well in practice.

Asynchronous Sequential Queries: We next explore another optimization that eliminates random disk I/Os completely from the second phase of query processing. The query processing algorithm in Figure 4 performs random I/Os because of Step [3] `P = P` $\cup$ `Descendants(`s_i`)` where sibling nodes on the left and right are alternately read after reading the initial leaf node. We can convert these random I/Os into sequential scans by maintaining extra information at the intermediate nodes. Specifically, with each pointer at an intermediate node, we maintain the size of the subtree (number of descendant points) hanging off that branch and a pointer to the rightmost left node in its level.

When descending the Prefix B-Tree in the first phase, we compare the size of the subtree with $y = max(c, m)$ at each branch. When the size of the branch b_z being taken becomes less than y for the first time, we step left over branches $b_{z-1}, b_{z-2}, \ldots, b_0$ until branch b when the sum of subtrees we have stepped over exceeds y or b is the left most branch of the tree. We then reach the least point in the subtree of b and perform a sequential scan collecting $2y$ points. It can be shown that all the y points which ASYNCHASCEND would have collected will be picked up in the sequential scan. Furthermore, since a sequential read is up to ten times faster than a random read, a sequential read of twice as many blocks as ASYNCHASCEND is still more efficient.

5.4 Parallelization

Both the main-memory and disk-based LSH Forest indexes can be parallelized to work over a large number of servers. For upto l servers, one option for parallelization is to simply place the l LSH Trees in round-robin order at participant servers.

A more interesting alternative – which works for any degree of parallelism – is to partition *each* LSH tree over all available nodes. The best way to do this partitioning is to use *range partitioning*, in which the points are sorted lexicographically by their LSH labels, and broken into contiguous ranges; each node is responsible for storing all the points falling in one particular range. Within each node, the set of points can be stored "independently" in an LSH Tree or Prefix B-Tree as discussed earlier. With such a range partitioning strategy, queries are easily parallelized and can usually be directed to just one node, helping achieve inter-query parallelism and linear speed-up. In fact, it is also possible to ensure efficient and dynamic load balancing across the different nodes, and even add or remove nodes on the fly [18].

5.5 Finding Similar Peers in a P2P System

A particularly interesting use of similarity search arises in peer-to-peer systems, where one may desire to find *peers* that are most similar to a given query, or peer. That is, each peer maps to a point, and we want to find the most similar peers to a given query. For example, each peer may correspond to a repository of documents, and a user may be interested in finding other peers similar to the one she is currently browsing, or similar to the specific document she is currently reading. In either case, we would like to use our LSH Forest index structure to enable efficient similarity search.

Of course, our notion of "efficient" in the P2P setting is very different from that in a single-disk system. We require the P2P similarity index to have two desiderata:

- The index should not be centralized; in fact, all peers should ideally perform an equal amount of work in handling queries, inserts and deletes.
- The number of messages exchanged between peers for queries as well as index maintenance (with peers joining and leaving the system) should be as small as possible.

It turns out that there is a natural embedding of LSH Forest in a P2P system that provides us with the above desiderata. We can exploit the fact that the objects being indexed are the peers themselves, and embed the LSH Forest in the *structure* of the overlay network connecting peers. Peers can open and maintain connections to other peers many of whom are similar to themselves, eliminating the need for an explicit index storage as described next:

Embedding the LSH Forest: Let us first consider how to embed a single LSH Tree in a P2P network. Recall that each peer (point) is a leaf in the LSH Tree and has a label of length $k \leq k_m$. For simplicity and w.l.o.g let us assume that each digit in the label is a single bit. We call the peer's label its *hostID,* and set up an overlay network connecting peers based on these hostIDs. The objective is to enable "emulation" of the logical operations on the prefix tree, while only having connections between the leaf nodes.

The overlay network is identical to an existing P2P structure called P-Grid [1], and is formed as follows: a peer p with hostID $l_1 l_2 \ldots l_k$ connects to a random peer p_i with whose hostID p shares exactly an i-length prefix, i.e., p_i has a hostID prefix $l_1 l_2 \ldots l_i \overline{l_{i+1}}$, for each $0 \leq i < k$. Intuitively, the peer can "jump" to its sibling branch at any level of the prefix tree using these links. We show next that all LSH Tree operations can be performed efficiently in this prefix-based network. The LSH Forest is constructed by simply forming l such overlay networks, one for each LSH tree.

Forest Maintenance in the presence of joins and leave of peers is performed by the regular P-Grid protocol [1] with no additional operations. It can be shown that peer joins and leaves require just $O(\log n)$ messages in an n-node overlay network. Of course, all peers need to be aware of the hash function sequence used to construct the different hostIDs, and have to apply the hash function to their own content in order to construct the hostIDs.

Queries: Recall that queries are answered by a bottom-up traversal of the different logical LSH Trees (synchronously or asynchronously), starting from a leaf that has the largest prefix match with the query. Consider a query q initiated by any peer P. For each LSH Tree, P starts out by hashing q to compute its label[4]; P then uses the overlay network's *routing mechanism* to find a peer P' that most closely matches q's label. (Note that, in the special case where a peer is looking for other peers most similar to itself, $P' = P$.) It can be shown that this routing process requires only $O(\log n)$ messages, no matter the shape of the LSH Tree [1].

In the case of synchronous query operation, all that remains to answer the query is specify how to find all peers which share a hostID prefix of at least x bits with P', for any value of x. (In case of asynchronous operation, what is required is a primitive that can find the c closest peers to P'. The rest of this discussion extends in an obvious fashion to this case.) It turns out that, given the link structure of the P-Grid, the number of messages used to find the y closest peers to P' is at most $y - 1$, for any value of y.

To see how, let us consider how P' can discover all peers with which it shares an x-bit prefix. Let P''s hostID be of length k, and let it maintain overlay-network links to a set of peers P_y, $x \leq y < k$, with P' and P_y sharing exactly a y-bit prefix. To collect all peers with which it shares at least an x-bit prefix, P' simply contacts each P_y and recursively requires P_y to discover all peers that share at least a $(y + 1)$-bit prefix with P_y.

Since all the P_ys themselves are part of the final list of nearest neighbors, we can see that each message produces at least one addition to the list of neighbors, showing that the total number of messages needed for finding y neighbors is at most $y - 1$. A query that needs to retrieve cl nearest neighbors from the l LSH Trees requires only $O(l(c + \log n))$ messages, irrespective of the shape of the LSH Trees, thus being efficient in communication costs.

6. EVALUATION METHODOLOGY

We implemented both the basic LSH scheme and the LSH Forest schemes (both SYNCHASCEND and ASYNCHASCEND) and studied their performance for similarity search in the text domain. We also evaluated the efficacy of the implementation for finding similar peers in a P2P system. We now describe the set-up of our evaluation, in terms of datasets, similarity functions, and LSH functions used, and quality metrics measured.

Datasets: The index structures are populated with one of the following two text data sets:

A TREC-1,2-AP: 506 MB of documents from AP Newswire in TREC CDs 1 and 2 comprising of 93,978 documents. Excluding documents without a valid author (for P2P experiments) led to 49,180 documents and 1,034 unique authors.

B Reuters: 798MB of news articles that comprise the Reuters Corpus (Vol. 1). The data set contains 124,010 documents and 1,568 unique authors.

For our standard indexing tests, the text of each news report was treated as a separate document to be indexed. For the P2P experiments, we associated each peer with all the documents written by a single author; thus, the objects being indexed were authors, with each author being treated as a giant document consisting of a concatenation of all news reports written by that author. All tests were run on all the datasets; where the results are similar, we present results on one of the datasets for brevity.

Terms: The documents are pre-processed to perform stemming and stop-word elimination. Each remaining term forms a distinct dimension in a high-dimensional space; each document is a vector in this space, with components along the terms present in that document. Each term in a document is weighted in proportion to its significance. We experimented with three weighting functions: TF (the number of occurrences of the term in the document), $\log TF$ (set to $1 + \log TF$ if a term occurs in the document, and 0 otherwise) and TF-IDF (set to $TF \log(N/DF)$ where DF is the number of documents in which the term occurs, and N is the total number of documents). Our experiments showed that the performance of indexing schemes is independent of the specific weighting function used; the experiments reported here all use $\log TF$.

As we will see momentarily, our similarity measures operate on sets of terms. We therefore convert the weighted vector into a set of terms by assigning each term a number of unique `termID`s according to its term weight. Thus an entry $< t_1, 3 >$ in a vector representing a term t_1 with weight 3 is (logically) represented as three `termID`s (t_1^1, t_1^2, t_1^3) in the set.

Similarity Metric: The degree of similarity between documents $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ is measured by a similarity function $sim(\overrightarrow{v_1}, \overrightarrow{v_2})$. Popular similarity functions for text include the Jaccard coefficient and Cosine similarity. The Jaccard coefficient treats vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ as sets of terms b_1 and b_2 (with weighted terms being represented by an appropriate number of distinct term IDs), and is defined as $[sim_J(b_1, b_2) = \frac{|b_1 \cap b_2|}{|b_1 \cup b_2|}]$. (The Jaccard distance is simply $1 - sim_J(b_1, b_2)$.) The Cosine similarity is the normalized inner product of the two vectors defined as $[sim_C(\overrightarrow{v_1}, \overrightarrow{v_2}) = \overrightarrow{v_1} \odot \overrightarrow{v_2}]$.

We will focus on the Jaccard metric in this work. Previous work has shown that the Jaccard metric yields comparable results on a Web crawl [22] or even outperforms Cosine similarity on an AP newswire corpus [26]. Furthermore, it turns out to be far easier

[4] The label can either be computed to length k_m in the beginning, or it can be computed lazily on demand.

to develop LSH families $\mathcal{H}$ for the Jaccard measure. Creating a corresponding family of LSH functions for the Cosine measure to support practical and efficient indexing is a hard problem [20].

LSH Functions $\mathcal{H}$ are used to hash document vectors in a locality-sensitive fashion. We use a locality sensitive technique for Jaccard distance measure called min-hashing [7] which works as follows. We begin with a family $\mathcal{H}$ of random linear functions (linear min-wise independent) of the form $h(x) = (ax + b) \mod p$. Each termID x_i in the document vector $\overrightarrow{v}$ is hashed using a function h drawn from $\mathcal{H}$, to obtain $h(x_i)$. Let $y = min_i\{h(x_i)\}$, that is, y is the minimum of the different termID hashes.

The value y is then hashed down to a single bit – 0 or 1 – using a second random hash function h' that maps each of $0, 1, 2, \ldots (p - 1)$ to 0 or 1. Thus, with two levels of hashing, each document is reduced to a single bit. (It can be shown that this hashing scheme is indeed locality sensitive for the Jaccard distance measure.) Of course, generating a k-bit signature entails the use of k different hash functions to produce one bit each.

Queries and Competing Algorithms: The index structures are used to answer top-m queries: given a query q, find m documents from the dataset that are most similar to q. The queries were obtained by picking random documents from the dataset. The following competing algorithms were used to find answers for the query:

LSH(K): The basic LSH scheme with fixed k in which candidate points are chosen at random from all points within the buckets to which query was hashed.

LSH S: The LSH Forest index in which candidate points are chosen synchronously across all LSH Trees.

LSH A: The LSH Forest index in which candidate points are chosen independently from each LSH Tree.

Random: As a frame of reference, a naive algorithm was designed that picks candidate points at random from the dataset.

Each algorithm picks as many as N candidate documents from the dataset. The candidates are ranked by computing their similarity with the query document, and the top m are returned as an answer.

Wherever possible, we also compare the results against an ORACLE that compares q against all documents in the dataset to pick the most similar documents (the *ideal* answer). Of course, such an oracular algorithm is too expensive to be feasible in practice.

Quality Measures: We measure the quality of an answer set A returned by algorithms by computing the *average similarity* of A to the query document, i.e., $sim(A, q) = \frac{\sum_{d \in A} sim(\overrightarrow{d}, \overrightarrow{q})}{|A|}$. We compare the "goodness" of A against an ideal answer I by computing the *relative error* defined as $err(A, I, q) = \frac{sim(I,q) - sim(A,q)}{sim(I,q)}$.

7. EMPIRICAL EVALUATION

In this section, we present results from our evaluation of LSH Forest on the above datasets, both for building a centralized index, as well as in the P2P setting. The focus of our study is on the *accuracy* of the query results obtained, and the *number* of candidates retrieved to achieve that accuracy. We do not present results on the number of disk accesses required since Prefix B-trees are well-understood indexing structures that have been studied for a number of years. Our simulations show the following results:

A The basic LSH scheme needs to be tuned carefully, based on the size and nature of the data being indexed, and the number of results desired.

B The LSH Forest consistently outperforms LSH(K), which was optimally tuned for a given static dataset and fixed number of desired results, by at least 15%.

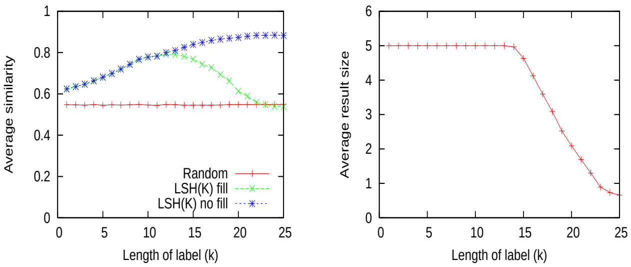

Figure 6: (a) The Jaccard coefficient of the top-5 peers from 25 candidate peers obtained using the LSH(K) algorithm. (b) The average number of results returned for the top-5 query.

C The LSH Forest outperforms LSH(K) by more than 33% when answering *ad hoc* queries with varying number of desired results (when retrieving twice the number of candidates as the number of desired results).

D The LSH A underperforms LSH S only by a slight margin, suggesting that the heuristic relaxation has good payoff.

E The P2P version of the LSH Forest is seen to return consistently high-quality results for all queries, while exploring fewer candidates, and exhibiting much lower variance in query result quality, as compared to the optimized LSH(K) scheme.

LSH Tree Structure

We start with a study of the structure of LSH Trees. Recall that our logical data structure is a prefix binary tree prone to imbalance caused by data skews. An imbalanced prefix tree will result in longer labels and increase the average insert, delete and query costs per tree. Furthermore, imbalances across LSH Trees can lead to slower synchronous bottom-up traversals.

Interestingly, all the LSH Trees that we built using our TREC and Reuters datasets were fairly well balanced. Figure 5(a) plots the average number of documents found at level i or below in a single LSH tree (Y-axis) for each value of i on a logarithmic scale (X-axis), using a set of $100,000$ documents from the Reuters corpus to populate the tree. Observe that there is a clear exponential decrease in the number of documents with increasing levels, suggesting that the tree is well-balanced.

Tuning The Basic LSH Index

Recall that the basic LSH scheme requires the label length k, and the number of indexes l as input parameters. We now study the effects of varying k on the behavior of the basic LSH scheme to highlight the importance of tuning k for good results. Figure 6(a) plots the average similarity of answers returned (Y-axis) against the corresponding k value (X-axis) for a top-5 query on the *Reuters* dataset with $l = 5$. The curve labeled `LSH(K) no fill` and `LSH(K) fill` plot the performance of two variants of the basic LSH scheme. The `LSH(K) no fill` scheme returns m best documents from n candidates for a top-m query. However, if it is only able to retrieve $m' < m$ candidates from its indexes, it only returns these m' documents. The `LSH(K) fill` scheme, on the other hand, always returns m documents in its answer. If $m' < m$, it picks $m - m'$ random documents from the dataset to complete its answer. Figure 6(b) plots the average number of answers returned by the `LSH(K) no fill` scheme (Y-axis) against the corresponding k values (X-axis).

We observe in Figure 6(a) that both `LSH(K) fill` and `LSH(K) no fill` schemes perform identically for small k values. The

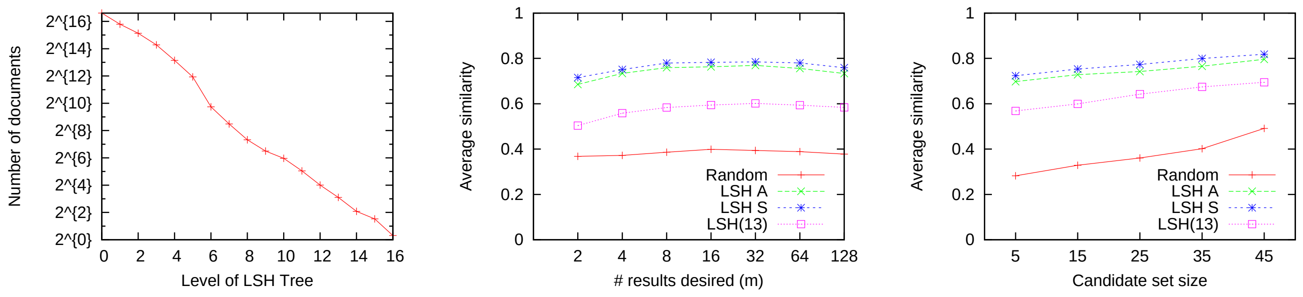

Figure 5: (a) The distribution of documents across levels in a LSH Tree. (b) The average similarity measures for LSH Forest and basic LSH for top-m queries. (c) The average similarity of top-5 queries with varying candidate set sizes.

schemes improve with increasing k, since there is a corresponding drop in collisions with distant points. As k increases beyond 13, the probability of collision decreases to the point where even nearby points stop colliding, and the number of collisions in a bucket drops below $m = 5$. The points that do lie in a bucket are highly similar, leading to a steady increase in the average similarity scores of `LSH(K) no fill`. However, the results returned begin to be less than 5 as shown by the knee in the curve of Figure 6(b). The `LSH(K) fill` responds by adding random points from the dataset, causing a drop in the average similarity score.

We note that the performance of basic LSH is critically determined by the value of k. If the k value is too small, the quality of results suffer. If the k value is too large, results are either incomplete or have poor quality. Furthermore, k depends on the underlying data distribution, the number of points n, and on the value of m. For example, if there are only 1000 documents in the data set, the optimal value of k shifts down to 8; similarly, a top-100 query would cause the "hump" in `LSH(K) fill` curve to shift left, requiring a low k value for optimal performance.

Comparing Optimal LSH(K) with LSH Forest

We are now ready to compare the competing algorithms: LSH Forest (`LSH S` and `LSH A`) with `LSH(K)`. We first consider the performance of the indexes for top-m queries for various values of m on the *Reuters* data set. We again use $l = 5$ as we discover that the incremental performance improvement from higher l values is very small. All schemes use a candidate set size of $M = 2m$. We use $k = 13$ for `LSH(K)`, which is optimal for top-5 queries as seen from the earlier experiments. For comparison, we also plot the performance of `Random`, which simply picks up M random points and returns the closest m of these, for comparison.

Figure 5(b) plots the average similarity measure for the Jaccard coefficient (Y-axis) against the value of m (X-axis). Note that the X-axis is on a log scale. We see that both LSH Forest schemes outperform LSH(K) by a large amount (around 33%) across the board. LSH S outperforms LSH A, but the margin is very small. We also see that there is a mild "hump" in all the curves: this is explained by the fact that, for low m, we are picking up too few candidates, some of which could be poor matches, while towards the end, the average similarity of the top m has to drop as m increases (since less and less similar peers are being added to the result set.)

Figure 5(c) compares the indexing schemes by plotting the average similarity for top-5 queries on *Reuters*, varying the number of query candidates on the x-axis. We see that as the candidate size varies from 5 to 45, there is a linear increase in the average similarity obtained by all the schemes. We do see however, that there is a significant gap of at least 15% in performance between the LSH Forest schemes and LSH(K) with an optimal k value.

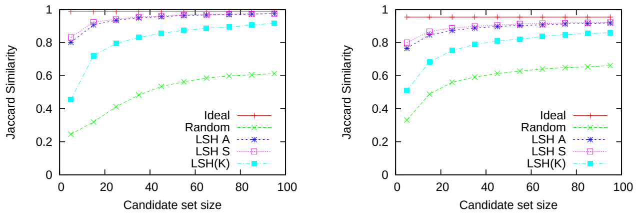

Figure 7: The average similarity measures for a top-5 query using (a) TREC and (b) Reuters to populate a P2P network.

P2P Experiments

We now consider the P2P setting where each peer is associated with all the news articles written by a single author, and similarity search is conducted over peers rather than over news articles. Our experiments focus on the accuracy of query results rather than on efficiency metrics for which we refer the interested reader to [1].

We repeated our experiments on basic LSH(K) to identify the optimal value of k for both TREC and Reuters data sets. (For the Reuters data, we found that the optimal value of k was 8 for top-5 queries, as opposed to the value 13 in the earlier $100,000$-document case.) Figures 7(a) and (b) plot the average similarity measures (Y-axis) against different candidate set sizes M (X-axis) for top-5 queries on TREC and Reuters data sets.

The LSH Forest schemes continue to perform better than the optimal LSH(K) scheme, even when LSH(K) picks up $5\times$ as many candidates. We do note, however, that LSH(K) receives a performance boost as the candidate set size increases. We also observe that the ideal similarity value is 0.97, and the LSH Forest schemes approach within 2% of ideal as the candidate set size increases.

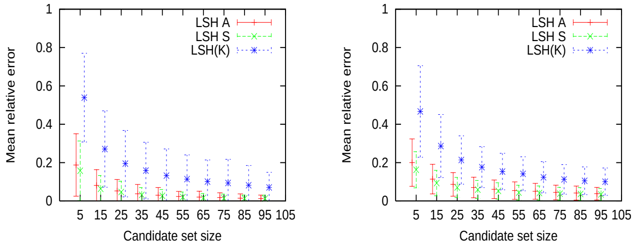

Figure 8: The mean relative error of basic LSH and LSH Forest schemes for top-5 queries on (a) TREC and (b) Reuters dataset.

We next designed an experiment in which *each* peer performed a top-5 query using its own site vector as the query point. Figures 8(a) and (b) plot the mean relative error for queries (Y-axis) against varying candidate set size M (Y-axis) for top-5 queries on both TREC and Reuters. The points represent the mean relative error, averaged over all queries, while the error bars reflect the 95% probability intervals for the error, i.e., 95% of the queries had relative errors within the plotted range.

The graphs clearly demonstrate that the LSH forest not only has lower relative error, but it also has *lower variance* in relative error. This means that it provides consistent performance for *all* queries. On the other hand, the LSH(K) scheme has both a higher relative error, and a higher variance in relative error, suggesting that *some queries perform poorly with LSH(K)*.

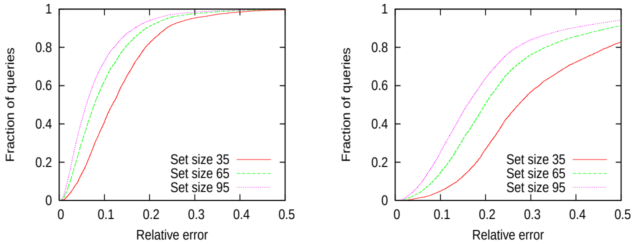

Figure 9: The distribution of each peer's relative error when using (a) `LSH A`, and (b) `LSH(K)`.

To reinforce this point further, Figure 9(a) plots the cumulative distribution of queries (Y-axis) against relative error (X-axis) for the Reuters data set observed for different candidate set sizes using `LSH A`. Figure 9(b) plots a similar graph for `LSH(K)`. We observe that even with a set size of 95, nearly 7% of queries in `LSH(K)` have a relative error greater than 0.5. On the other hand, with the same candidate set size, practically no queries using `LSH A` have a relative error greater than 0.3. This conclusively demonstrates the advantages of LSH Forest over `LSH(K)` in providing accurate answers for *all* queries.

8. CONCLUSIONS

We have presented a self-tuning index for similarity search called LSH Forest. The LSH Forest can be applied for constructing main-memory, disk-based, parallel and peer-to-peer indexes for similarity search. We have shown that the LSH Forest improves on the basic LSH scheme in terms of the accuracy of the results returned and, perhaps more importantly, by eliminating the need for various data-dependent tuning parameters. In consequence, we have developed a practical plug-and-play solution for similarity indexing that only requires an LSH-compatible similarity function as input.

9. REFERENCES

[1] K. Aberer. Scalable data access in p2p systems using unbalanced search trees. In *Proc. WDAS*, 2002.

[2] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Transactions on Database Systems*, 2(1), 1977.

[3] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. of SIGMOD*, 1998.

[4] S. Berchtold, D. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. of VLDB*, 1996.

[5] K. Bharat and A. Broder. Mirror, mirror, on the web: A study of host pairs with replicated content. In *Proc. of WWW*, 1999.

[6] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. of SIGMOD*, 1995.

[7] A. Broder. On the resemblance and containment of documents. In *Proc. of Compression and Complexity of Sequences*, 1998.

[8] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proc. of WWW*, 1997.

[9] S. Chakrabarti, B. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, and J. Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proc. of WWW*, 1998.

[10] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *Proc. of SIGMOD*, 2000.

[11] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proc. of ICDE*, 2000.

[12] B. Cooper, N. Sample, M. Franklin, and M. Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, 2001.

[13] B. Davison. Topical locality in the web. In *Proc. of SIGIR*, 2000.

[14] J. Dean and M. Henzinger. Finding related pages in the world wide web. In *Proc. of WWW*, 1999.

[15] C. Faloutsos and K.-I. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. of SIGMOD*, 1995.

[16] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proc. of VLDB*, 1998.

[17] P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2), 1999.

[18] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proc. VLDB*, 2004.

[19] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Proc. of Hypertext and Hypermedia*, 1998.

[20] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of VLDB*, 1999.

[21] A. Gutman. R-trees: A dynamic index structure for spatial searching. In *Proc. of SIGMOD*, 1997.

[22] T. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proc. of WWW*, 2002.

[23] P. Indyk and R. Motwani. Approximate nearest neighbor - towards removing the curse of dimensionality. In *Proc. of STOC*, 1998.

[24] N. Katayama and S. Satoh. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of SIGMOD*, 1997.

[25] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proc. of SODA*, 1998.

[26] L. Lee. Measures of distributional similarity. In *Proc. of ACL*, 1999.

[27] D. Lomet. The evolution of effective b-tree: Page organization and techniques: A personal account. *SIGMOD Record*, 30(3), 2001.

[28] E. McCreight. Pagination of b*-trees with variable-length records. *Communications of the ACM*, 20(9), 1977.

[29] D. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[30] J. Robinson. The k-d-b tree: A search structure for large multidimensional indexes. In *Proc. of SIGMOD*, 1981.

[31] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proc. of ACM DL*, 1996.

[32] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity search methods in high dimensional spaces. In *Proc. VLDB*, 1998.

[33] D. White and R. Jain. Similarity indexing with ss-tree. In *Proc. of ICDE*, 1996.