Streaming Similarity Search over One Billion Tweets Using Parallel Locality-Sensitive Hashing

Streaming Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing

Narayanan Sundaram[†], Aizana Turmukhametova [⋆], Nadathur Satish[†], Todd Mostak[⋆], Piotr Indyk[⋆], Samuel Madden[⋆] and Pradeep Dubey[†]

[†]Parallel Computing Lab, Intel
{narayanan.sundaram}@intel.com

[⋆]CSAIL, MIT
{aizana,tmostak,indyk,madden}@mit.edu

ABSTRACT

Finding nearest neighbors has become an important operation on databases, with applications to text search, multimedia indexing, and many other areas. One popular algorithm for similarity search, especially for high dimensional data (where spatial indexes like kd-trees do not perform well) is Locality Sensitive Hashing (LSH), an approximation algorithm for finding similar objects.

In this paper, we describe a new variant of LSH, called Parallel LSH (PLSH) designed to be extremely efficient, capable of scaling out on multiple nodes and multiple cores, and which supports high-throughput streaming of new data. Our approach employs several novel ideas, including: cache-conscious hash table layout, using a 2-level merge algorithm for hash table construction; an efficient algorithm for duplicate elimination during hash-table querying; an insert-optimized hash table structure and efficient data expiration algorithm for streaming data; and a performance model that accurately estimates performance of the algorithm and can be used to optimize parameter settings. We show that on a workload where we perform similarity search on a dataset of > 1 Billion tweets, with hundreds of millions of new tweets per day, we can achieve query times of 1–2.5 ms. We show that this is an order of magnitude faster than existing indexing schemes, such as inverted indexes. To the best of our knowledge, this is the fastest implementation of LSH, with table construction times up to $3.7\times$ faster and query times that are $8.3\times$ faster than a basic implementation.

1. INTRODUCTION

In recent years, adding support to databases to identify similar objects or find nearest neighbors has become increasingly important. Hundreds of papers have been published over the past few years describing how to extend databases to support similarity search on large corpuses of text documents (e.g., [24]), moving objects (e.g., [14]), multimedia (e.g., [12]), graphs (e.g., [32]), genetic sequences (e.g., [23]), and so on.

Processing such queries is a challenge, as simple attempts to evaluate them yield linear algorithms that compare the query object to every other object. Linear algorithms are particularly unattractive when the objects being compared are complex or multi-dimensional, or when datasets are large, e.g., when comparing trajectories of a large number of moving objects or computing document similarity measures over dynamic text corpuses (e.g., Twitter).

Many efficient nearest neighbor algorithms have been proposed. For low-dimensional data, spatial indexing methods like kd-trees [11] work well, but for higher dimensional data (e.g., vector representations of text documents, images, and other datasets), these structures suffer from the "curse of dimensionality", where there are many empty hypercubes in the index, or where query performance declines exponentially with the number of dimensions. In these cases, these geometric data structures can have linear query times. For such higher dimensional data, one of the most widely used algorithms is locality sensitive hashing (LSH) [20, 13]. LSH is a sub-linear time algorithm for near(est) neighbor search that works by using a carefully selected hash function that causes objects or documents that are similar to have a high probability of colliding in a hash bucket. Like most indexing strategies, LSH consists of two phases: *hash generation* where the hash tables are constructed, and *querying*, where the hash tables are used to look up similar documents. Previous work has shown that LSH is the algorithm of choice for many information retrieval applications, including near-duplicate retrieval [19, 25], news personalization [15], etc.

In this work, we set out to index and query a corpus of approximately 1 billion tweets, with a very high update rate (400 million new tweets per day) [6]. Initially we believed that existing LSH algorithms would provide a straightforward means to tackle this problem, as tweets are naturally modeled as high-dimensional objects in a term-vector space. Surprisingly, applying LSH to very large collections of documents, or document sets that change very frequently, proved to be quite difficult, especially in light of our goal of very fast performance that would scale to thousands of concurrent queries (of the sort Twitter might encounter). This can be ascribed to the following factors:

1. Modern processors continue to have more and more cores, and it is desirable to take advantage of such cores. Unfortunately, obtaining near-linear speedups on many cores is tricky, due to the need to synchronize access of multiple threads to the data structure.

2. Existing variants of LSH aren't distributed. Because LSH is a main-memory algorithm, this limits the maximum corpus size.

3. Additionally, hash table generation and querying in LSH can become limited by memory latency due to cache misses on the irregular accesses to hash tables. This means that the application no longer utilizes either compute or bandwidth resources effectively.

4. High rates of data arrival require the ability to efficiently expire old data from the data structure, which is not a problem that previous work has addressed satisfactorily.

5. The LSH algorithm takes several key parameters that determine the number of hash tables and number of buckets in each hash table. Setting these parameters incorrectly can yield exces-

Proceedings of the VLDB Endowment, Vol. 6, No. 14

sive memory consumption or sub-optimal performance, and prior work provides limited guidance about how to set these parameters.

To address these challenges, we developed a new version of LSH, which we call "Parallel LSH" (PLSH), which we describe in this paper. In order to get PLSH to perform well, we needed to make a number of technical and algorithmic contributions such as:

1. Both hash table construction and hash table search in PLSH are distributed across multiple cores and multiple nodes.

2. Within a single node, multiple cores concurrently access the same set of hash tables. We develop techniques to batch and rearrange accesses to data to maximize cache locality and improve throughput.

3. We introduce a novel cache-efficient variant of the LSH hashing algorithm, which significantly improves index construction time. We also perform software prefetching and use large pages to minimize cache miss effects.

4. To handle streaming arrival and expiration of old documents, we propose a new hybrid approach that buffers inserts in an insert-optimized LSH delta table and periodically merges these into our main LSH structure.

5. We develop a detailed analytical model that accurately projects the performance of our single- and multi-node LSH algorithms to within 25% of obtained performance. This model allows us to estimate the optimal settings of PLSH parameters on modern hardware and also allows us to understand how close our observed performance on modern systems is to expected performance.

Although LSH has been widely used, as we note in Section 2, previous work has not shown how to optimize LSH for modern multi-core processors, and does not include the optimizations for efficient parallel operation on LSH tables. We believe PLSH is quite general, and should efficiently support many of the previous applications of LSH mentioned above [19, 25, 15].

Taken together, versus an unoptimized implementation, our optimizations improve hash table construction times by a factor of $3.7\times$, and query times by a factor of $8.3\times$ on a 1 Billion Twitter data set, with typical queries taking 1–2.5 ms. In comparison to other text search schemes, such as inverted indexes, our approach is an order of magnitude faster. Furthermore, we show that this implementation achieves close to memory bandwidth limitations on our systems, and is hence bound by architectural limits. We believe this is the fastest known implementation of LSH.

2. RELATED WORK

LSH is a popular approach for similarity search on high-dimensional data. As a result, there are numerous implementations of LSH available online, such as: E2LSH [1], LSHKIT [4], LikeLike [2], LSH-Hadoop [3], LSH on GPU [27] and Optimal LSH [5]. Among these, LikeLike, LSH-Hadoop and LSH-GPU were designed specifically for parallel computational models (MapReduce, Hadoop and GPU, respectively). LikeLike and LSH-Hadoop are distributed LSH implementations; however they do not promise high performance. LSH-GPU, on the other hand, is oriented towards delivering high levels of performance but is unable to handle large datasets because of current memory capacity limitations of GPUs. However, to the best of our knowledge, none of these implementations have been designed for standard multi-core processors, and are unable to handle the large scale real-time applications considered in this paper.

There have been many variations of LSH implementations for distributed systems to reduce communication costs through data clustering and clever placement of close data on nearby nodes e.g. [18]. Performing clustering however requires the use of load balancing techniques since queries are directed to some nodes but not others. We show that even with a uniform data distribution, the communication cost for running nearest neighbor queries on large clusters is $< 1\%$ with little load imbalance. Even with other custom techniques for data distribution (e.g., [10]), each node eventually runs a standard LSH algorithm. Thus a high performing LSH implementation that achieves performance close to hardware limits is a useful contribution to the community that can be adopted by all.

Our paper introduces a new cache-friendly variant of the all-pairs LSH hashing presented in [7], which computes all LSH hash functions faster than a naive algorithm. There are other fast algorithms for computing LSH functions, notably the one given [16] that is based on fast Hadamard transform. However, the all-pairs approach scales linearly with the sparsity of the input vectors, while the fast Hadamard transform computation takes at least $\Omega(D \log D)$ time, where D is the input dimension (which in our case is about $500,000$). As a result, our adaptation of the all-pairs method is much more efficient for the applications we address.

The difficulty of parameter selection for the LSH algorithm is a known issue. Our approach is similar to that employed in [31] (cf. [1]), in that we decompose the query time into separate terms (hashing, bucket search, etc.), estimate them as a function of the parameters k, L, δ (see Section 3) and then optimize those parameters. However, our cost model is much more detailed. First, we incorporate parallelism into the model. Second, we separate the cost of the computation, which depends on the number of *unique* points that collide with the query, from the cost that is a function of the *total* number of collisions. As a result, our model is very accurate, predicting the actual performance of our algorithm within a 15-25% margin of error.

LSH has been previously used for similarity search over Twitter data [28]. Specifically, the paper applied LSH to Twitter data for the purpose of first story detection, i.e. those tweets that were highly dissimilar to all preceding tweets. In order to reduce the query time, they compare the query point to a constant number of points that collide with the query most frequently. This approach works well for their application, but it might not be applicable to more general problems and domains. To extend their algorithm to streaming data, they keep bin sizes constant and overwrite old points if a bin gets full. As a result, each point is kept in multiple bins, and the expiration time is not well-defined. Overall, the heuristic variant of LSH introduced in that work is accurate and fast enough for the specific purpose of detecting new topics. In this paper, however, we present an efficient general implementation of LSH that is far more scalable and provides well-defined correctness guarantees. In short, our work introduces a high performance, in-memory, multithreaded, distributed nearest neighbor query system capable of handling large amounts of streaming data, something no previous work has achieved.

3. ALGORITHM

Locality-Sensitive Hashing [20] is a framework for constructing data structures that enables searching for near neighbors in a collection of high-dimensional vectors. The data structure is parameterized by the *radius* R and *failure probability* δ. Given a set $\mathcal{P}$ containing D-dimensional input vectors, the goal is to construct a data structure that, for any given query q, reports the points within the radius R from q. We refer to those points as *R-near neighbors* of q in $\mathcal{P}$. The data structure is randomized: each R-near neighbor is reported with probability $1 - \delta$ where $\delta > 0$. Note that the correctness probability is defined over the random bits selected by the algorithm, and we do not make any probabilistic assumptions about the data set.

The data structure utilizes *locality sensitive* hash functions. Consider a family $\mathcal{H}$ of hash functions that map input points to some

universe U. Informally, we say that $\mathcal{H}$ is *locality-sensitive* if for any two points p and q, the probability that p and q collide under a random choice of hash function depends only on the distance between p and q. We use the notation $p(t)$ to denote the probability of collision between two points within distance t. Several such families are known in the literature, see [8] for an overview.

In this paper we use the locality-sensitive hash families for the *angular distance* between two unit vectors, defined in [13]. Specifically, let $t \in [0, \pi]$ be the angle between unit vectors p and q. t can be calculated as $\text{acos}(\frac{p \cdot q}{||p|| \cdot ||q||})$. The hash functions in the family are parametrized by a unit vector a. Each such function h_a, when applied on a vector v, returns either -1 or 1, depending on the value of the dot product between a and v. Specifically, we have $h_a(v) = \text{sign}(a \cdot v)$. Previous work shows [13] that, for a random choice of a, the probability of collision satisfies $p(t) = 1 - t/\pi$. That is, the probability of collision $P[h_a(v_1) = h_a(v_2)]$ ranges between 1 and 0 as the angle t between v_1 & v_2 ranges between 0 and π.

Basic LSH: To find the R-near neighbors, the basic LSH algorithm concatenates a number of functions $h \in \mathcal{H}$ into one hash function g. In particular, for k specified later, we define a family $\mathcal{G}$ of hash functions $g(v) = (h_1(v), \ldots, h_k(v))$, where $h_i \in \mathcal{H}$. For an integer L, the algorithm chooses L functions $g_1, \ldots, g_L$ from $\mathcal{G}$, independently and uniformly at random. The algorithm then creates L hash arrays, one for each function g_j. During preprocessing, the algorithm stores each data point $p \in \mathcal{P}$ into buckets $g_j(v)$, for all $j = 1, \ldots, L$. Since the total number of buckets may be large, the algorithm retains only the non-empty buckets by resorting to standard hashing.

To answer a query q, the algorithm evaluates $g_1(q), \ldots, g_L(q)$, and looks up the points stored in those buckets in the respective hash arrays. For each point p found in any of the buckets, the algorithm computes the distance from q to p, and reports the point v if the distance is at most R.

The parameters k and L are chosen to satisfy the requirement that a near neighbor is reported with a probability at least $1-\delta$. See Section 7 for the details. The time for evaluating the g_i functions for a query point q is $O(DkL)$ in general.

For the angular hash functions, each of the k bits output by a hash function g_i involves computing a dot product of the input vector with a random vector defining a hyperplane. Each dot product can be computed in time proportional to the number of non-zeros NNZ rather than D. Thus, the total time is $O(\text{NNZ}kL)$.

All-pairs LSH hashing: To reduce the time to evaluate functions g_i for the query q, we reuse some of the functions $h_j^{(i)}$ in a manner similar to [7]. Specifically, we define functions u_i in the following manner. Suppose k is even and $m \approx \sqrt{L}$. Then, for $i = 1 \ldots m$, let $u_i = (h_1^{(i)}, \ldots, h_{k/2}^{(i)})$, where each $h_j^{(i)}$ is drawn uniformly at random from the family $\mathcal{H}$. Thus u_i are vectors each of $k/2$ functions (bits, in our case) drawn uniformly at random from the LSH family $\mathcal{H}$. Now, define functions g_i as $g_i = (u_a, u_b)$, where $1 \leq a < b \leq m$. Note that we obtain $L = m(m-1)/2$ functions g_i.

The time for evaluating the g_i functions for a query point q is reduced to $O(Dkm + L) = O(Dk\sqrt{L} + L)$. This is because we need only to compute m functions u_i, i.e., mk individual functions h, and then just concatenate all pairs of functions u_i. For the angular functions the time is further improved to $O(\text{NNZ}km + L)$. In the rest of the paper, we use "LSH" to mean the faster all-pairs version of LSH.

In addition to reducing the computation time from $O(DkL)$ to $O(Dk\sqrt{L} + L)$, the above method has significant benefits when combined with 2-level hashing, as described in more detail in Section 5.1.2. Specifically, the points can be first partitioned using the first chunk of $k/2$ bits, and the partitioning can be refined further using the second chunk of $k/2$ bits. In this approach the first level partitioning is done only m times, while the second level hashing is done on much smaller sets. This significantly reduces the number of cache misses during the table construction phase.

To the best of our knowledge, the cache-efficient implementation of the all-pairs hashing is a new contribution of this paper.

Hash function for Twitter search: In the case of finding nearest neighbors for searching Twitter feeds, each tweet is represented as a sparse high-dimensional vector in the vocabulary space of all words. See Section 8 for further description.

4. OVERALL PLSH SYSTEM

In this work, we focus on handling nearest neighbor queries on high-volume data such as Twitter feeds. In the case of Twitter feeds, there is a stream of incoming data that needs to be inserted into the LSH data structures. These structures need to support very fast query performance with very low insertion overheads. We also need to retire the oldest data periodically so that capacity is made available for incoming new data.

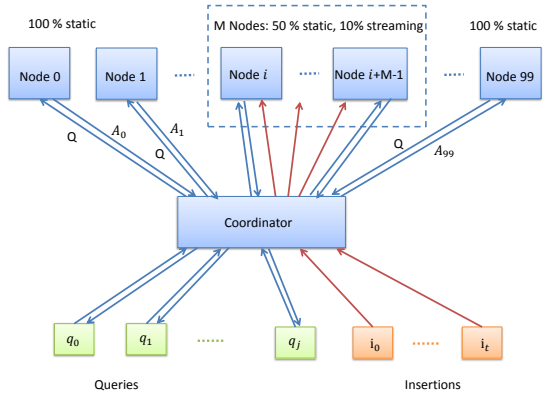

Figure 1: Overall system for LSH showing queries and insert operations. All nodes hold part of the data and participate in queries with a coordinator node handling the merging of query results. For inserts, we use a rolling window of M nodes (Node i to $i + M - 1$) that have a streaming structure to handle inserts in round-robin fashion. These streaming structures are periodically merged into static structures. The figure shows a snapshot when the static structures are 50% filled. When they are completely filled, the window moves to nodes $i + M$ to $i + 2M - 1$.

Figure 1 presents an overview of how our system handles high-volume data streams. Our system consists of multiple nodes, each storing a portion of the original data. We store data in-memory for processing speed; hence the total capacity of a node is determined by its memory size. As queries arrive from different clients, they are broadcast by the coordinator to all nodes, with each node querying its data. The individual query responses from each structure are concatenated by the coordinator node and sent back to the user.

PLSH includes a number of optimizations designed to promote efficient hash table construction and querying. First, in order to speed up construction of static hash tables (that are not updated dynamically as new data arrives) we developed a 2-level hashing approach, that when coupled with the all-pairs LSH method described in Section 3 significantly reduces the time to construct hash

tables. Rather than using pointers to track collisions on hash buckets this approach uses a partitioning step to exactly allocate enough space to store the entries in each hash bucket. This algorithm is described in Section 5.1. We show that this algorithm speeds up hash table construction by up to a factor of $3.7\times$ versus a basic implementation.

In order to support high throughput parallel querying, we also developed a number of optimizations. These include: i) processing the queries in small batches of a few hundred queries, trading latency for throughput, ii) a bitmap-based optimization for eliminating duplicate matches found in different hash tables, and iii) a technique to exploit hardware prefetching to lookup satisfying matches once item identifiers have been found. These optimizations are described in Section 5.2. In Section 8, we show that our optimized query routine from can perform static queries as fast as 2.5 ms per query, a factor of $8.3\times$ faster than a basic implementation.

We have so far described the case when there are no data updates. In Section 6 we describe how we support updates using a write-optimized variant of LSH to store *delta tables*. Queries are answered by merging answers from the static PLSH tables and these delta tables. Since queries to the delta tables are slower than those to static tables, we periodically merge delta tables into the static tables, buffering incoming queries until merges complete. Deletions are handled by storing the list of deleted entries on each node; these are eliminated before the final query response.

Taken together, these optimizations allow us to bound the slowdown for processing queries on newly inserted data to within a factor of 1.5X, while easily keeping up with high rates of insertions (e.g., Twitter inserts 4600 updates/second with peaks up to 23000 updates/second) [6].

5. STATIC PLSH

In this section, we describe our high-performance approach to construction and querying of a static read-only LSH structure consisting of many LSH tables. Most data in our system described in Section 4 is present in static tables, and it is important to provide high query throughput on these. We also describe in Section 6 how our optimized LSH construction routine is fast enough to support fast merges of our static and delta structures.

We focus on a completely in-memory approach. Since memory sizes are increasing, we have reached a point where reasonably large datasets can be stored across the memory in different nodes of a multi-node system. In this work, we use 100 nodes to store and process more than a billion tweets.

The large number of tables (L is around 780 in our implementation) and the number of entries in each table (k can be 16 or more, leading to at least 2^k = 64K entries per table) leads to performance bottlenecks on current CPU architectures.

Specifically, there are two performance problems that arise from a large number of hash tables: (1) time taken to construct and query tables linearly scales with the number of tables – hence it is important to make each table insertion and query fast, and (2) when performing a single query, the potential neighbor list obtained by concatenating the data items in the corresponding hash bins of all hash tables contains duplicates. Checking all duplicates is wasteful, and these duplicates need to be efficiently removed.

In addition, unoptimized hash table implementations can lead to performance problems both during table construction and querying due to multi-core contention. Specifically, in LSH, nearby data items, by design, need to map to similar hash entries, leading to hash collisions. Existing hash table implementations that employ direct chaining and open addressing methods [22] have poor cache behavior and increased memory latency. Array Hashes [9] increase memory consumption and do not scale to multiple cores.

These considerations drove the techniques we now describe for producing an optimized, efficient representation of the static representation of hash tables in PLSH (we describe how we support streaming in Section 6.)

We first define a few symbols for ease of explanation.
N : Number of data items.
D : Dimensionality of data.
k : Number of bits used to index a single hash table.
L : Number of hash tables used in LSH.
m : Number of $k/2$-bit hash functions used. Combinations of these are used to generate $L = m(m-1)/2$ total k-bit hashes.
T : Number of hardware threads (including SMT/Hyperthreading).
S : SIMD width.

5.1 LSH table construction

Given a choice of LSH parameters k and L (or equivalently, m), the two main steps in LSH table construction are (1) hashing each data point using each hash function to generate the k-bit indices into each of the L hash tables, and (2) inserting each data point into all L hash tables. All insertions and queries are done using data indexes 0...N-1 (these are local to each node, and hence we assume them to be 4-byte values).

In this section we describe the approach we developed in PLSH, focusing in particular on the new algorithms we developed for step (2) that involve a two-level hierarchical merge operations that works very well in conjunction with the all pairs LSH-hashing method described in Section 3.

5.1.1 Hashing data points

Recall that (as described in Section 3) the LSH algorithm begins by computing $m*k/2$ hash functions that compute angular distances. Each hash function is obtained as a dot-product between the sparse term vector representing the tweet and a randomly generated hyperplane in the high-dimensional vocabulary space. Due to the use of these angular distances, evaluating the hash functions over all data points can be treated as a matrix multiply, where the rows of the first matrix stores the data to be hashed in sparse form (IDF scores corresponding to the words in each tweet), and the columns of the second matrix store the randomly generated hyperplanes. The output of this is a matrix storing all hash values for each input. In practice, the first matrix is very sparse (there are only about 7.2 words per tweet out of a vocabulary of 500000), and hence is best represented as a sparse matrix. We use the commonly used Compressed Row Storage (CRS) format [17] for matrix storage.

Parallelism: These steps are easily parallelized over the data items N, yielding good thread scaling. Note that it is possible to structure this computation differently by iterating instead over columns of the dense (hyperplane) matrix; however structuring the computation as parallelization over data items ensures that the sparse matrix is read consecutively, and that at least one row of the dense matrix is read consecutively. This improves the memory access pattern, allowing for SIMD computations (e.g., using Intel® AVX extensions).

There can be cache misses while accessing the dense matrix since we can read rows that are widely apart. Such effects can in general be reduced by using a blocked CRS format [17]. However, in our examples, the Zipf distribution of words found in natural languages lead to some words being commonly found in many tweets. Such words cause the corresponding rows of the dense matrix to be accessed multiple times, which are hence likely to be cached.

This behavior, combined with the relatively large last level caches on modern hardware (20 MB on Intel® Xeon® processor E5-2670) results in very low cache misses. In practice, we see less than 10% of the memory accesses to the dense matrix miss the last level cache (LLC). Hence this phase is compute bound, with efficient use of compute resources such as multiple cores and SIMD.

5.1.2 *Insertion into hash tables*

Once the m $k/2$-bit hashes have been computed, we need to create L k-bit hashes for each tweet and insert each tweet into the L hash tables so generated. In this section, we describe our new two-level partitioning algorithm for doing this efficiently. They idea is that we want to construct hash tables in memory that are represented as contiguous arrays with exactly enough space to store all of the records that hash to (collide in) each bucket, and we want to do this in parallel and in a way that maximizes cache locality.

Given all $k/2$-bit hashes $u_1,...u_m$ for a given tweet, we take pairs of hashes (u_i, u_j), and concatenate them into k-bit hashes $g_{i,j}$. There are $L = \binom{m}{2}$ such combinations of hashes we can generate. Each such hash $g_{i,j}$ maps a tweet to a number between 0 and $2^k - 1$, and this tweet can then be inserted into a specific bucket of a hash table. A naïve implementation of this hash table would have a linked list of collisions for each of the 2^k buckets. This leads to many problems with both parallel insertion (requiring locks), and irregular memory accesses.

Given a static (non-dynamic) table, we can construct the hash table in parallel using a contiguous array without using a linked list. This process of insertion into the hash tables can be viewed as a **partitioning** operation. This involves three main steps: first, scan each element of the table and generate a histogram of entries in the various hash buckets; (2) perform a cumulative sum of this histogram to obtain the starting offsets for each bucket in the final table, and (3) perform an additional scan of the data and recompute the histogram, this time adding it to the starting offsets to get the final offset for each data item. This computation needs to be performed for each of the L hash tables. Note that a similar process has been used for hash-join and radix-sort, where it was also shown how to parallelize this process [21, 30].

However, although easily parallelized, we found the performance to be limited by memory latency – especially when the number of buckets is large. One common problem of partitioning algorithms is the limited size of the Translation Lookaside Buffer (TLB) which is used to perform virtual to physical address translation on modern hardware. As previously described by Kim et al. [21], for good partitioning performance, the number of partitions should not be more than a small multiple of the first level TLB size (64/thread in Intel® Sandy bridge). In practice, PLSH typically requires k to be about 16, and having 2^{16} = 64K partitions results in significant performance loss due to TLB misses.

In order to improve performance, we perform a **2-level hierarchical partitioning** similar to a Most-Significant Bit (MSB) radix sort. Given the k-bit hash, we split it into two halves of $k/2$ bits each. We first partition the data using the first $k/2$ bits, resulting in $2^{k/2}$ *first-level* partitions. Each first-level partition is then independently rearranged using the second $k/2$ bit-key. For example, the hash table for $g_{1,2}$ can be first partitioned on the basis of the first half of the key (u_1), thus creating $2^{k/2}$ partitions. Each such partition is then partitioned by u_2. Similarly, the table for $g_{1,3}$ can be partitioned by u_1 and u_3 respectively. The main advantage is that each of these steps only creates $2^{k/2}$ partitions at a time (256 for k=16), which results in few TLB misses.

In the case of LSH, this 2-level division matches precisely with the way the hashes are generated. This leads to opportunities to share the results of the first-level partitions among the different hash functions. For instance, consider the hash tables for $g_{1,2} = (u_1, u_2)$ and $g_{1,3} = (u_1, u_3)$. Both these tables are partitioned first using the same hash function u_1. Instead of repeating this partitioning work, we can instead **share the first level partitioned table** among these hashes. Figure 2 and Table 1 show an example of this sharing. In order to achieve this, we again use a 2-step process: we first partition the data on the basis of the m hash functions $u_1..u_m$. This step only involves m partitions. In the second step, we take each first level partition, say u_i, and partition it on the basis of $u_{i+1}...u_m$. This involves a total of $L = \binom{m}{2}$ partitions. This reduces the number of partitioning steps from $2 * L$ (first and second level partitions) to $L+m$, which takes much less time since $m \sim O(\sqrt{L})$.

	u_1	u_2	u_3	u_4
$t1$	10	11	11	00
$t2$	00	00	10	00
$t3$	00	11	01	11
$t4$	10	11	11	10
$t5$	11	11	10	00
$t6$	11	10	10	10
$t7$	10	10	10	01
$t8$	10	11	00	00
$t9$	10	01	11	01
$t10$	00	10	01	10

Table 1: Example with $k = 4$, $m = 4$, $L = 6$: hashing of 10 datapoints wi

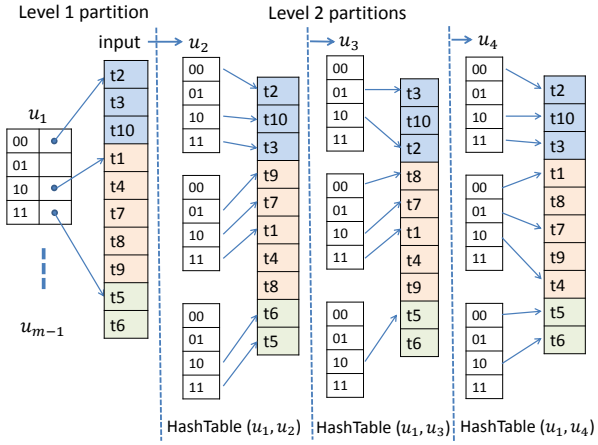

Figure 2: Example with $k = 4$, $m = 4$, $L = 6$: sharing of the first-level partition among the different hash functions. Hash tables (u_1, u_2), (u_1, u_3) and (u_1, u_4) are first partitioned according to u_1. Each partition (shown in different colors) is then partitioned according to hash functions u_2, u_3 and u_4. The corresponding hash values are shown in Table 1.

Detailed Algorithm: The following operations are performed :
Step I1: Partition all data items (rather their indices initialized to 0..N-1) according to each of the m $k/2$ bit hashes. This is done using the optimized three-step partitioning algorithm described earlier in this section. Also store the final scatter offsets into an array.
Step I2: Create all hashes required for the second level partition. For each final hash table $l \in 1..L$ corresponding to a first level hash function g_i and a second level function g_j, rearrange the hash values $g_j(n), n \in 1..N$ according to the final scatter offsets created in Step I1c for g_i. These will reflect the correct order of hashes for the permuted data indexes for g_i.
Step I3: Perform second level partitions of the permuted first-level data indexes using the hashes generated in Step I2. This follows the same three step procedure as in Step I1. This is done for a total of L hash tables.

Of these steps, Step I1 only does rearrangement for m hash functions and takes O(m) time. Steps I2 and I3 work with all L hash tables, and thus dominate overall runtime of insertion.

Parallelism: Step I1 is parallelized over data items, as described

in [21]. Threads find local histograms for their data, and then go into a barrier operation waiting for other threads to complete. One of the threads performs the prefix sum of all local histogram entries of all threads to find per-thread offsets into the global output array. The threads go into another barrier, and compute global offsets in parallel and scatter data. Step I2 is also parallelized over data items for each hash table. Finally, Step I3 has additional parallelism over the first level partitions. Since these are completely independent, we parallelize over these. To reduce load imbalance, we use the task queueing [26] model, whereby we assign each partition to a task and use dynamic load balancing using work-stealing.

Insertions into the hash table are mostly limited by memory bandwidth, and there is little performance to be gained from using SIMD.

5.2 Queries in PLSH

In this section, we discuss the implementation used while querying the static LSH structures. As each query arrives, it goes through the following steps:

Step Q1: The query is hashed using all $m*k/2$ hash functions and the hash index for each of the L hash tables is computed.

Step Q2: The matching data indices found in each of the hash tables are merged and duplicate elimination is performed to determine the unique data indexes.

Step Q3: Each unique index in the merged list is used to look up the full data table, and the actual distance between the query and the data item is computed (using dot products for Twitter search)

Step Q4: Data entries that are closer to the query than the required radius R are appended to an output list for the query.

Step Q1 only performs a few operations per query, and takes very little time. Step Q4 also generally takes very little time since very few data items on average match each query and need to be appended to the output. Most time is spent in Steps Q2 and Q3.

Parallelism: All Steps are parallelized over queries that are completely independent. To reduce the impact of load imbalance across different queries, we use work-stealing task queues with each query being a task. In order to achieve sufficient parallelism for multi-threading and load-balance, we **buffer** at least 30 queries and process them together, at the expense of about 45 ms latency in query responses. We benchmark optimization in Section 8.

We next describe three key optimizations that PLSH employs to speed up querying. These are: (i) An optimized bitvector representation for eliminating duplicates in Step Q2; (ii) A prefetching technique to mask the latency of RAM in step Q3; and (iii) A representation of queries as sparse bit vectors in the vocabulary space to optimize the computation of dot products in Step Q3.

5.2.1 Bitvector optimization to merge hash lookups

The first optimization occurs in Step Q2, which eliminates duplicate data indexes among the data read from each hash table. There are three basic ways this could be done: (1) by sorting the set of duplicate data items and retaining those data items that are not the same as their predecessors, (2) using a data structure such as a set to store non-duplicate entries using an underlying structure such as red-black trees or binary search trees, or (3) using a histogram to count non-zero index values. The first and second methods involve $O(Q \log Q)$ operations over the merged list containing duplicates Q. If the indices in the hash buckets were maintained in sorted order, then we could do (1) using merge operations rather than sorts. However, even then, we are sorting L lists of length Q/L (say), which will take $O(Q \log L)$ time overall using merge trees. The third technique can be done in constant time per data index or $O(Q)$ overall, with a small constant. Specifically, for each data index, we can check if the histogram value for that index is 0, and if so write out the value and set the histogram to 1, and if not skip that index.

The choice of (2) vs (3) has some similar characteristics to the more general sort vs. hash debate for joins and sorts [21, 30]. Since our data indices fall in a limited range (0..N-1), we can use a **bitvector to store the histogram**. For N = 10 million, we only need about 1.25 MB to store the histogram. Even with multiple threads having private bitvectors, we can still keep them **in cache** given that modern processors have about 20 MB in the last level cache. Computing this bit-vector is bound by compute resources. Hence, we use histograms and bitvectors for duplicate elimination.

5.2.2 Prefetching data items

The bitvector described above stores ones for each unique data index that must be compared to the query data. For each such one, the corresponding tweet data (identifying which words are present in the tweet and their IDF scores), has to be loaded from the data tables and distances from the query computed. We first focus on the loading of data and then move on to the actual computation.

We find that the access of data items suffers from significant memory latency. There are two main reasons for this: (1) the set of data items to be loaded is small and widely spread out in memory, which results in misses to caches and TLB (2) the hardware prefetcher fails to load data items into cache since it is difficult to predict which items will be loaded.

To handle the first problem, we use large 2 MB pages to store the actual data table to store more of the data in TLB (support for 1 GB pages is also available and can completely eliminate these misses – we did not find this necessary in this work). However, it is difficult to solve the prefetch problem if we only use bit-vectors to store unique indexes. Fundamentally, given a bit set to 1, the position of the next bit set to 1 is unpredictable.

In order to get around this problem, we scan the bitvector and store the non-zero items into a separate array. Note that this array is inherently sorted and only has unique elements. We use this array to identify succeeding data items to prefetch – when computing distances for one data item, we issue software prefetches to get the data corresponding to succeeding data items into cache. A linear scan of the bit-vector can use SIMD operations to load the bits and can use a lookup table to determine the number of ones in the loaded data. Although this has to scan all bits, in practice the time spent here is a small constant. This operation is also CPU-bound.

5.2.3 Performing final filtering

Once data is loaded into cache, the distance between the data item and query must be computed. For Twitter search, this distance is a dot product between two sparse vectors – one representing a data item and the other a query. Each dimension in the vector represents the IDF score for a word in the tweet. Each sparse vector is stored using a data array (containing IDF scores) and an index array (showing which word in the vocabulary is contained in the tweet).

One approach to find this sparse dot-product is to iterate over the data items of one sparse vector, and perform a search for the corresponding index in the other sparse vector's index array. If a match is found, then the corresponding IDF scores are multiplied and accumulated into the dot-product. In order to efficiently perform this computation, we form a sparse **bit-vector in the vocabulary space** representing the index array of the query, and use O(1) lookups into this bit-vector to decide matches. Note that this bit-vector is different from the one used to find unique data indexes from the hash tables – that bit-vector is over the set of data indexes 0...N-1 rather than the vocabulary space. The query bit-vector is small

(only 500K bits) and comfortably fits in L2 cache. In practice, the number of matches is very small (only around 8% of all bit-vector checks result in matches), and hence the computation time mainly involves fast O(1) lookups. It turns out that the overall time for Step Q3 (load data item and compute the sparse dot-product) is limited by the memory bandwidth required to load the data. We provide more details in Section 7.

5.3 Multi-Node PLSH

The main motivation to use multiple nodes for LSH is to scale the system capacity to store more data. Given a server with 64 GB DDR3 memory, and using N = 10 million tweets and typical LSH parameters $L = 780$ ($m = 40$), the total size of the LSH tables is given by $L * N * 4$ bytes = 31 GB. Along with additional storage required for the actual data plus other structures, we need about 40 GB memory. This is nearly two-thirds of our per-node memory. In order to handle a billion tweets, we need about a hundred nodes to store the data.

There are two ways to partition the data among the nodes. First, each node could hold some of the L hash tables across all data points. Second, each node could hold all hash tables but for a subset of the total data N. The first scheme suffers two problems (1) it incurs heavy communication costs since unique elements have to be found globally across the hash entries of different nodes (Step Q2 in Section 5.2); (2) L is a parameter depending on desired accuracy and does not scale with data size N. It is possible for L to be less than the number of nodes, and this will limit node scaling. Therefore we adopt the second scheme in this work.

Since each node stores part of the data, LSH table constructions and queries occur on the data residing in each node in parallel. In our scheme, we evenly distribute the data in time order across the nodes, with nodes getting filled up in round-robin order as data items arrive. As queries arrive, they are broadcast to all nodes, with each node producing a partial result that is concatenated. It is possible to think of alternative methods that attempt to perform clustering of data to avoid the communication involved in broadcasting queries. However, we show in Section 8 that this broadcast takes well under 1% of overall time, and hence the overheads involved in data clustering plus potential load-balancing issues will be too high to show benefits for these techniques. We also show in Section 8 that the load imbalance across nodes for typical query streams is small, and that query performance is constant with increasing node counts while keeping the data stored per node constant.

6. HANDLING STREAMING DATA

In real life applications, such as similarity search in Twitter, the data is streaming. There are, on average, around 400 million new tweets per day, equating to approximately about 4600 tweets per second [6]. Static LSH is optimized for querying, and insertion of new data requires expensive restructuring of the hash tables. Therefore, we propose a new approach where we buffer inserts in *delta tables* to handle high rates of insertions efficiently. These delta tables are stored using an insert-optimized variant of LSH that uses dynamic arrays (vectors) to accommodate the insertions. As a consequence, queries on the delta tables are slower than on the optimized static version. (The details of the delta table implementation are given in Section 6.1 below.)

Upon the arrival of a query from the user, we query both static and delta tables and return the combined answer. When the number of points in the delta table is sufficiently low, the query runs fast enough. Once the delta table hits a threshold of a fraction η of the overall capacity C of a node, its content is merged into the static data structure. The fraction η is decided such that the query performance does not drop by more than an empirical bound of 1.5X from that of static queries. This is a worst case bound; and only happens when the delta structure is nearly full.

When the total capacity of the node is reached, old data needs to be retired for new data to be inserted. In a multi-node setting, there are many possible policies we can use to decide how retirement happens. This is closely tied to insertion policies. Under the assumption that streaming data is uniformly inserted to all nodes (in a round robin manner), it is very difficult to identify old data without the overhead of keeping timestamps. One approach that could be used is to use circular queues to store LSH buckets, overwriting elements when buckets overflow **??**. In this scenario, there is no guarantee that the same data item is deleted from all buckets; this can also affect accuracy of results. We adopt an alternative approach where we can easily and gracefully identify old data to be retired. Consider our system in Figure 1. In our system, we limit insertions to a set of M nodes (M is smaller than the total number of nodes) at a time in round-robin fashion. Initially, all updates go to the first set of M nodes, and we move on to the next set when these nodes get full. This continues until all nodes have reached their capacity. At this point, new data insertions require some old data to be retired. The advantage of our system is that we know that the first set of M nodes store the oldest data, and hence can be retired (the contents of the these nodes are erased). Insertions then begin to the delta tables as in the beginning. Note that at any point, all nodes except possibly for M will be at their peak capacity.

The main drawback to our system is that all updates go to only M nodes at a time. We must choose M to ensure that updates and merges are processed fast enough to meet input data rates. In practice, with our optimized insertion and merge routines, we find that M = 4 is sufficient to be able to meet Twitter traffic rates with overheads of lower than 2%.

6.1 Delta table implementation

Delta tables must be able to support two conditions: they must be able to support fast queries while also supporting fast insertions. There are at least 2 ways to implement such a streaming structure. A commonly used structure is a simple linear array which is appended to as data items arrive. This is easy to update, but queries require a linear scan of the data. This leads to unacceptably poor performance – e.g., a 2× slow down with only η =1% of the data in the delta table. Hence we do not pursue this approach.

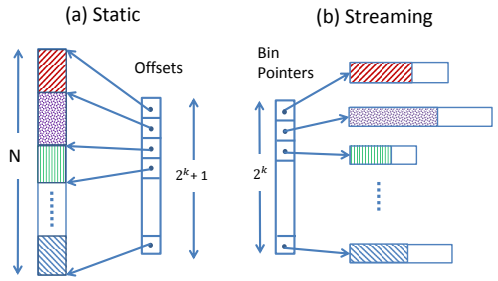

Figure 3: Comparison of (a) static and (b) streaming LSH tables. Static tables are contiguous and optimized for read access. Streaming (or delta) tables support fast inserts using dynamically reallocated arrays for each bucket. However, query performance is slower on these structures, requiring periodic merges with static structures for good performance.

The second approach involves maintaining hash tables as in PLSH – with the exception that each bucket has to be dynamically updatable. We retain the same parameter values (k, L) as for the static LSH data structures (although it is technically possible to have different values). The main difference between the static and streaming LSH structures is how the data is arranged. As mentioned in

Section 5, static LSH has a set of L arrays, each with N points partitioned according to their hash values. For delta tables, we use a streaming variant of LSH that has a set of $2^k \times L$ resizeable vectors. Every new tweet is hashed and inserted into L of these bins. These insertions can be done independently for each table, allowing us the exploit multiple threads to parallelize the computation. Figure 3 illustrates this difference between an optimized static hash table and a delta table. Using hash table based delta tables allows us to easily integrate the results of static and streaming structures without any linear scans of data. We find that query performance is only slightly affected (less than 1.5X – Section 8 has details) for delta structures as large as η =10% of overall data. We show insertion performance for different delta table sizes in Section 8.

6.2 Low overhead updates

In order to maintain query performance on our overall system, it is important to keep the overheads of updates (insertions, deletions and merges) as low as possible. Our overall LSH system (Section 4) ensures all inserts (and hence subsequent merges) go to the delta tables of M nodes at a time. As M increases, these inserts and merges are done in parallel over more nodes; hence reducing their overhead. However, as we increase M, we retire more data simultaneously, hence temporarily reducing system capacity. In practice, we do not want M to be larger than 4 or so (out of 100), and this drives our performance requirements on inserts and merges.

Insertions: We first consider insertions. We process insertions into the delta table are processed in batches of about 100K. This allows us to amortize insertion costs, but means that the newest data will not be reflected in query results until 100K updates arrive at each of M nodes. For M=4 and at Twitter update rates of 4600 updates/second, this occurs in about 86 seconds.Using a batch of this size, our LSH delta tables allow us to process 100K updates in around 400 milliseconds (Section 8), which parallelized across M=4 nodes, is under 100 milliseconds. This overhead is incurred roughly every 86 seconds. Inserts thus take about 0.4% of the overall time.

Merging of Delta and Static Tables: We next consider merges. The merge step combines static hash tables and delta tables into one static data structure. This must be done once delta tables grow large enough that query performance slows significantly. One way to perform the merge is simply to reinitialize the static LSH structure, but with the streamed data added. We can easily show that although this is unoptimized, no merge algorithm can be more than 3X better. This is seen by noting that our initialization time is bound by memory bandwidth, with traffic of about 32 bytes per entry in the combined tables (Section 7). Any other merge scheme will at least have to read the static table and write the combined tables, taking about 12 bytes of traffic. Hence our bound of 32/12=2.67X.

In practice, our merge costs are around 15 seconds in the worst case when the static and delta buffers are both full (from construction costs for static LSH in Section 8). The merge time is nearly independent of the size of the delta array (since most data is assumed to be in the static array). Hence it is advantageous to have larger delta buffers to reduce the frequency of merge overheads. For η = 10% of overall capacity, merges happen once 1 Million inserts accumulate at a node (C=10M). This happens every 864 seconds for Twitter with M=4. Hence merge overhead is about 1.7%. Queries received during the merge are buffered until the merge completes.

Deleting Entries: Deletions of arbitrary tweets can be handled through the use of a bitvector similar to that used for eliminating duplicates in Section 5.2. Before performing the sparse dot product computation, we check this bitvector to see if the corresponding entry is "live" and proceed accordingly. This bitvector gets reset to all-zeros when the data in the node is retired. This does not add significantly to query times due to the low overhead of a single bitvector access.

6.3 Worst-case vs Average case query rates

The overall query processing time is the sum of the query times on the static and delta table LSH structures. We ignore update costs since they are very low.

In the following discussion, we will express the distribution of data among static and delta tables as a tuple (p_S, p_D), where p_S and p_D represent the fraction of overall capacity C in the static and delta tables. We assume p_D is bounded by a number η, after which merges are performed. It is important to note that the nodes involved in the insertion of data items may not be at peak capacity, hence $p_S + p_D$ is less than 1.

All nodes except the ones performing inserts have all their data in static LSH tables, and that the static tables are full, corresponding to a data distribution of (1.0, 0.0). For the nodes performing inserts, the **worst case** query times occur when both the static table as well as delta tables are nearly full, corresponding to a distribution of $((1-\eta), \eta)$. We want to size η such that this performance is no worse than 1.5X that of static LSH. For 10M entries, query time for static LSH =1.4 ms and for streaming LSH = 6 ms (Section 8), giving us $\eta \leq 0.15$. We choose $\eta = 0.1$.

Note that nodes are not always in this worst-case state. Most of the time, the static tables on nodes where inserts happen are not full. We show in Section 8 that in the average case when the static tables are 50% full, i.e. (0.50, 0.10), query times are slightly lower than 1.4 ms. Hence in many average case scenarios, the performance of static LSH on the non-inserting nodes is actually the bottleneck!

7. PERFORMANCE MODEL

In this section, we present a hardware-centric performance model for the core components of LSH namely, time taken to insert a tuple into a hash table, and time taken to query (retrieve a single entry from the hash table, find unique entries from the set of retrieved entries, filter the unique entries). We then discuss the overall performance model. These models are important because they allow us to determine the optimal setting of PLSH parameters on different hardware. We show in Section 8 that this model is quite accurate.

7.1 Model for fundamental operations

We first provide a model for LSH query, and then for LSH initialization. We will use the LSH query model to provide a framework for selecting LSH parameters in the remainder of this section. We use LSH initialization model in Section 6 while discussing streaming merge operations.

7.1.1 LSH query

LSH query consists of four main steps detailed as Q1-Q4 in Section 5.2. Of these, Steps Q1 (hashing queries) and Steps Q4 (writing out neighbors within the input radius to the output array) take little time and are ignored. Step Q2 involves (1) reading the indexes from the hash tables and forming a bit-vector to remove duplicates among them, and (2) scanning this bit-vector and writing out the non-zeros to an array for ease of later prefetching. Reading the data indexes involves 4 bytes of memory traffic per index read. On our part, we obtain around 12.3 bytes/cycle (around 32 GBps achieved bandwidth and 2.6 GHz frequency). The bandwidth limit is 4/12.3 = 0.3 cycles per index.

We now describe computation requirements. To update the bitvector, first the specific 32-bit word within which the bit is to updated has to be computed (2 operations), the word read (2 ops), the

specific bit extracted (shift & logical and - 2 ops), checked with 0 (2 op) and then if it is zero, set the bit (2 ops). Further, there is a loop overhead of about 3 operations, leading to a total average of around 11 ops per index. Performing about 1 operation per cycle, this takes 11 cycles per index. With 8 cores, this goes down to 11/8 = 1.4 cycles per index. Further, scanning the bit-vector consumes about 10 ops per 32-bits of the bit-vector to load, scan and write the data, and another 4 ops for loop overheads. Hence this takes about 14/8 = 1.75 cycles per 32-bits of N, or 0.6M cycles for N=10M. This is independent of number of indexes read. Thus **Step Q2** requires a total of T_{Q2} = **1.4 cycles/duplicated index + 0.6M cycles**, and is compute bound.

Step Q3 requires the reading of the tweet data for the specific data items to be compared to the tweet. This is only required for the unique indexes obtained in Step Q2. However, each data item read involves high bandwidth consumption of around 4 cache lines or about 256 bytes. To understand this, consider that transfers between the memory and processor caches are done in units of cache lines (or 64 bytes), where entire lines are brought into cache even if only part of it is accessed. In our case, there are three data structures accessed (CRS format [17]). Of these, two data loads for a given tweet are typically only 30 bytes (half a cache line). However, they are not aligned at cache line boundaries. In case the start address is in the last 30 bytes of the 64 bytes, then the access crosses a cache line – requiring 2 caches line reads. On average, each of the arrays requires 1.5 cache lines read, hence a total of 4 along with the cache line from the third array. These 256 bytes of read result in 256/12.3 = 20.8 cycles per data item accessed. Hence T_{Q3} **= 21.8 cycles/unique index (or call to sparse dot product)**.

7.1.2 LSH initialization

LSH initialization involves two main steps - hashing the input data and insertion into the hash tables. As per Section 5.1, hashing is compute intensive, and performs operations for each non-zero element. The total cost is equal to N*NNZ, where the average number of non-zeros NNZ $\sim$ 7.2 for Twitter data. For each such element, steps H1 and H2 are performed with each of the m*$k/2$ hash values for that dimension. Each such operation involves load of the hash (2 ops), multiply (1 ops), load output value (2 ops), add to output (1 op), store output (2 ops). In addition, about 3 operations are required for handling looping. Thus a total of 11 ops for each hash and non-zero data item combination. These operations parallelize well (11/8 ops in parallel), and also vectorize well (close to 8X speedup using AVX), resulting in (11/8/8) ops/hash/non-zero data. For k=16 and m=40, and assuming one cycle per operation, **hashing takes** a total of T_H **= 412 cycles/tweet**.

Insertion into the hash table itself involves three steps - Steps I1-I3 (Section 5.1). Step I1 involves reading of each hash twice (8 bytes), reading the data index (4 bytes), writing out the rearranged data indexes (4 bytes) and writing the offsets (4 bytes). Each write also involves a read for cache coherence, hence total memory traffic of 8 + 4*2 + 4*2 = 24 bytes per data item per first-level hash table. This phase is bandwidth limited, with a performance of T_{I1} = 24/12.3 = 1.96 cycles * m = 1.96m cycles/tweet. With m=40, this is around 78 cycles/tweet. Step I2 involves creating each hash used in second-level insert. For each of the L hash tables obtained using pairs of hash functions (u_i, u_j), computing bandwidth requirements shows that 16 bytes of traffic is required. Step I3 performs second level insertions into the L hash tables, and takes another 16 bytes traffic per table. For m=40, L=780, hence T_{I2} = T_{I3} = 16*780 /12.3 = 1015 cycles/tweet. Hence total **insertion time is** T_I **=** T_{I1} **+** T_{I2} **+** T_{I3} **= 2108 cycles/tweet**.

Total construction takes T_H + T_I = 2520 cycles/tweet. More than 80% of the time is spent in steps T_{I2} and T_{I3}, which are bandwidth limited with about 32 bytes of traffic per tweet per hash table. Section 8 shows that this is fast enough to use in streaming merge.

7.2 Overall PLSH performance model

Here, we put things together with the overall LSH algorithm parameters such as (number of entries expected to be retrieved from the hash tables, number of unique entries) to give the overall model. This uses the k, L (or m),R parameters together with the fundamental operation times above.

We describe how to select the parameters required by the LSH algorithm. First, we set the failure probability δ (i.e., the probability of not reporting a particular near neighbor) to 10%. This guarantees that a vast majority (90%) of near neighbors are correctly reported. As seen in Section 8, this is a conservative estimate - in reality the algorithm reports 92% percent of the near neighbors.

Second, we choose the radius R. Its value is tailored to the particular data set, to ensure that the points within the angular distance R are indeed "similar". We have determined empirically that for the Twitter data set the value $R \approx 0.9$ satisfies this criterion.

Given R and δ, it remains to select the parameters k and L of the LSH algorithm, to ensure that each R-near neighbor is reported with probability at least $1 - \delta$. To this end, we need to derive a formula that computes this probability for given k and L (or m).

Fast LSH Consider a query point q and a data point v. Let t be the distance between q and v, and $p = p(t)$. We need to derive an expression for the probability that the algorithm reports a point that is within the distance R from the query point. With functions g_i, the point is not retrieved if q and v collide on only *zero* or *one* of the functions u_i. The probability of the latter event is equal to

$$P'(t,k,m) = 1 - \left(1 - p(t)^{k/2}\right)^m - m \cdot p(t)^{k/2} \cdot \left(1 - p(t)^{k/2}\right)^{m-1}$$

The algorithm chooses k and m such that $P'(R,k,m) \geq 1 - \delta$.

7.3 Parameter selection

The values of k, L are chosen as a function of the data set to minimize the running time of a query while ensuring that each R-near neighbor is reported with probability $1 - \delta$. Specifically, we enumerate pairs (k, m) such that $P'(R,k,m) \geq 1 - \delta$, and for each of the pair we estimate the total running time.

We decompose the running time of the query into 4 components as mentioned in Section 5.2. Of the 4 steps, steps Q2 and Q3 dominate the runtime, so we focus on those components only. Step Q2 concatenates the indices of the points found in all L buckets, and determines the unique data indexes. This takes time T_{Q2} · #collisions, where #collisions is the number of collisions of points and queries in all of the L buckets. Note that a point found in multiple buckets is counted *multiple* times.

The expected value of #collisions for query q is

$$E[\#\text{collisions}] = L \cdot \sum_{v \in \mathcal{P}} p^k(\text{distance}(q,v)) \qquad (7.1)$$

Step Q3 uses the unique indices to look up the full data table, and computes the actual distance between the query and the data item. This takes time T_{Q3} · #unique where #unique is the number of unique indices found. The expected value of #unique for q is

$$E[\#\text{unique}] = \sum_{v \in \mathcal{P}} P'(\text{distance}(q,v), k, L) \qquad (7.2)$$

In summary, the parameters k and m (and therefore $L = m(m-1)/2$) are selected such that the expression

$$T_{Q2} E[\#\text{collisions}] + T_{Q3} E[\#\text{unique}]$$

is minimized, subject to

$$P'(R, k, m) \geq 1 - \delta \tag{7.3}$$

$$(L \cdot N + 2^k \cdot L) * 4 \leq \text{Memory in bytes} \tag{7.4}$$

This can be done by enumerating $k = 1, 2, \ldots k_{max}$, and for each k selecting the smallest value of m satisfying Equation 7.3. The values of E[#unique] and E[#collisions] can be estimated from a given dataset using equations 7.1 and 7.2 through sampling. We use a random set of 1000 queries and 1000 data points for generating these estimates.

For small amounts of data, we can set k_{max} to 40, as for larger k the collision probability for two points within distance R would be less than $p(t)^k = 0.71^{40} \leq 10^{-6}$. For large amounts of data, k_{max} is determined by the amount of RAM in the system. The storage required for the hash tables increases with L, which in turn increases super-linearly with k. The $L \cdot N$ term in Equations 7.4 dominates memory requirements. For 10 million points in a machine with 64GB of main memory, we can only store 1600 hash tables (excluding other data structures). Typically, we want to store about 1000 hash tables. This fixes the maximum value of m to be 44 and the largest k to satisfy Equation 7.3 then is 16. Enumeration can proceed as earlier and the best value of (k, m) is chosen.

8. EVALUATION

We now evaluate the performance of our algorithm on an Intel® Xeon® processor E5-2670 based system with 8 cores running at 2.6 GHz. Each core has a 64 KB L1 cache and a 256 KB L2 cache, and supports 2-way Simultaneous Multi-Threading (SMT). All cores share a 20MB last level cache. Bandwidth to main memory is 32 GB/s. Our system has 64 GB memory on each node and runs RHEL 6.1. We use the Intel® Composer XE 2013 compiler for compiling our code[1]. Our cluster has 100 nodes connected through Infiniband (IB) and Gigabit Ethernet (GigE). All our multi node experiments use MPI to communicate over IB.

Benchmarks and Performance Evaluation: We run PLSH on 1.05 billion tweets collected from September 2012 to February 2013. These tweets were cleaned by removing non-alphabet characters, duplicates and stop words. Each tweet is encoded as a sparse vector in a 500,000-dimensional space corresponding to the size of the vocabulary used. In order to give more importance to less common words, we use an Inverse Document Frequency (IDF) score that gives greater weight to less frequently occurring words. Finally we normalize each tweet vector to transform it into a unit vector. For single node experiments, we use about 10.5 million tweets. The optimal LSH parameters were selected using our performance model. We use the following parameters: $k = 16, m = 40, L = 780, D = 500,000, R = 0.9, \delta = 0.1$.

For queries, we use a random subset of 1000 tweets from the database. 0-length queries are possible if the tweet is entirely composed of special characters, unicode characters, numerals, words that are not part of the vocabulary etc. Since these queries will not find any meaningful matches, we ignore these queries. Even though we use a random subset of the input data for querying, we have found empirically that queries generated from user-given text snippets perform equally well.

8.1 PLSH vs exact algorithms

Algorithm	# distance computations	Runtime
Exhaustive search	10,579,994	115.35 ms
Inverted index	847,027.9	$>$ 21.81 ms
PLSH	120,345.7	1.42 ms

Table 2: Runtime comparison between PLSH and other deterministic nearest neighbor algorithms. Inverted index excludes time to generate candidate matches, only including the time for distance computations.

In order to prove empirically that PLSH does indeed perform significantly better than other algorithms, we perform a comparison against an exhaustive search and one based on an inverted index. The exhaustive search algorithm calculates the distance from a query point to all the points in the input data and reports only those points that lie within a distance R from the query. An inverted index is a data structure that works by finding a set of documents that contain a particular word. Given a query text, the inverted index is used to get the set of all documents (tweets) that contain at least one of the words in the document. These candidate points are filtered using the distance criterion. Both the exhaustive search and inverted index are *deterministic* algorithms. LSH, in contrast, is a randomized algorithm with a high probability of finding neighbors.

Table 2 gives the average number of distance computations that need to be performed for each of the above mentioned algorithms for a query (averaged from a set of 1000 queries) and their runtimes on 10.5 million tweets on a single node. Since we expect that all these runtimes are dominated by the data lookups involved in distance computations, it is clear that PLSH performs much better than both these techniques. For inverted index, we do not include the time to generate the candidate matches (this would involve lookups into the inverted index structure), whereas for PLSH we include all times (hash table lookups and distance calculations). Even assuming lower bounds for inverted index, PLSH is about $15\times$ faster than inverted index and $81\times$ faster than exhaustive search while achieving 92% accuracy. Note that all algorithms have been parallelized to use multiple cores to execute queries.

8.2 Effect of optimizations

We now show the breakdown of different optimizations performed for improving the performance of LSH as described in Section 5. Figure 4 shows the effect of the performance optimizations applied to the creation (initialization) of LSH, as described in Section 5.1. The unoptimized version performs hash table creation using a separate k-bit key for each of the L tables. Starting from an unoptimized implementation (1-level partitioning), we achieve a $3.7\times$ improvement through the use of our new 2-level hash table and optimized PLSH algorithm, as well as use of shared hash tables and code vectorization. All the versions are parallelized on 16 threads.

Figure 5 provides a breakdown of the effects of the optimizations on LSH query performance described in Section 5.2. The series of optimizations applied to the unoptimized implementation include the usage of bitvector for removing duplicate candidates, optimizing the sparse dot product calculation, enabling prefetching of data and the usage of large pages to avoid TLB misses. The unoptimized implementation uses the C++ STL set to remove duplicates and uses the unoptimized sparse dot calculation (Section 5.2.3). Compared to this, our final version gives a speedup of $8.3\times$.

8.3 Performance Model Validation

In this section, we show that the performance model proposed in Section 7 corresponds closely to the real world performance of PLSH creation and querying. Figure 6 compares the estimated and real runtimes of PLSH. Some of the error comes from the estimates of E[#collisions] and E[#unique] through sampling and other errors from inaccurately modeling the PLSH component kernels. We

[1] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

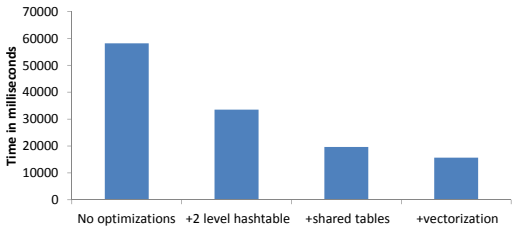

Figure 4: PLSH creation performance breakdown. The optimizations are described in Section 5.

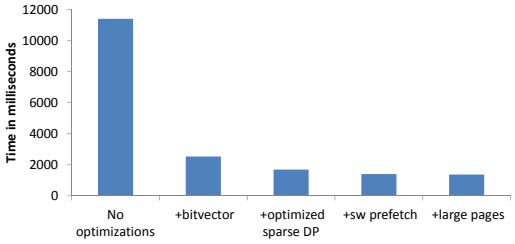

Figure 5: PLSH query performance breakdown (for 1000 queries). The optimizations are described in Section 5.

demonstrate the performance model accuracy on an additional text dataset obtained from 8 million Wikipedia article abstracts with a 500K vocabulary. We find that the error margin is less than 15% for both PLSH creation and querying on Twitter data and less than 25% for the Wikipedia data. Even more importantly, we can accurately estimate relative performance changes correctly when parameters change. Figure 7 demonstrates that our performance model can be used for tuning LSH parameters (for $R = 0.9, \delta = 0.1$). As mentioned earlier, given a main memory Twitter and Wikipedia data respectively. In all cases, our model provides reliable estimates of both relative and absolute performance numbers. These results suggest that our model can be used both to estimate whether a given hardware configuration can manage a particular query/insert load and also to optimize parameter selection.

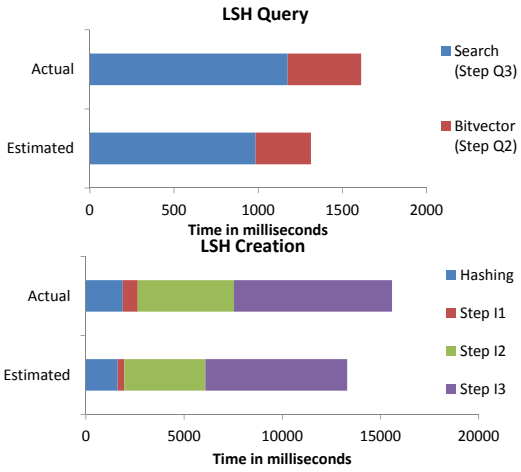

Figure 6: Estimated vs actual runtimes for PLSH creation & querying (1000 queries).

8.4 Scaling

In this section, we discuss the scaling performance of PLSH. First, Figure 8 shows how the performance of PLSH improves on a single 8-core node with increasing number of threads. We see that we achieve a speedup of 7.2× for PLSH initialization and 7.8× for querying.These performance numbers have already been shown to be very close to the achievable bandwidth and compute limits of the hardware.

Figure 9 shows the performance of PLSH as we add more data and nodes to our problem. Here, we keep the amount of work per processor constant and increase the number of processors (and correspondingly, the amount of data). This demonstrates that our implementation is indeed scalable to more than a billion tweets without any load balancing or communication problems. We define load balance as the ratio of the maximum to average runtime. Results indicate that this ratio is smaller than 1.3 (ideal is 1.0) for both PLSH creation and querying even at the largest scale. We also found that

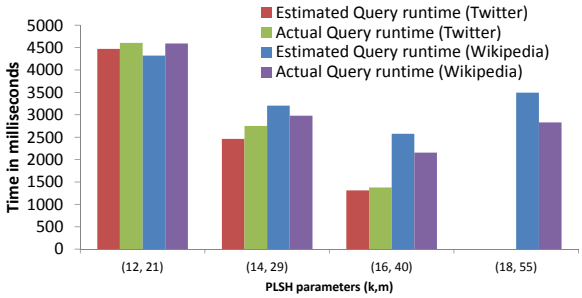

Figure 7: Estimated vs actual runtimes for varying k and m (1000 queries)

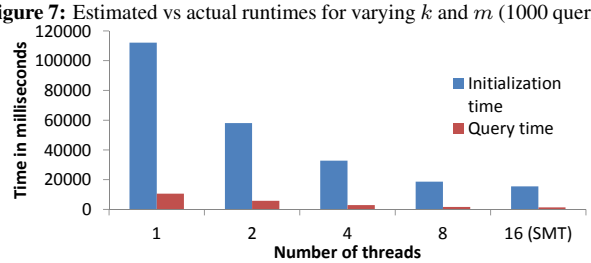

Figure 8: Scaling performance with increasing threads on a single node.

the query communication time is less than 20 ms in all cases (less than 1% of overall runtime).

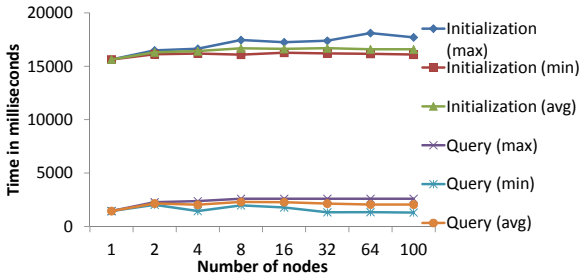

Figure 9: Scaling performance on multiple nodes. The data per node is fixed at 10.5 million tweets, so flat lines indicate perfect scaling.

8.5 Latency vs Throughput

So far in this section, we have measured query performance on a set of 1000 queries. We do this for 2 reasons. (1) We believe that while performance for an isolated query is important, most analytics workloads would need to support multiple queries at the same time. Techniques like flu tracking using Twitter [29] rely on detecting several concepts i.e. searching for several terms/concepts. (2) Using multithreading to process multiple queries works well in practice compared to multithreading a single query search. Tight synchronization is needed for processing a single query whereas multiple independent queries can be processed simultaneously without the need for synchronization or locks.

Figure 10 shows the latency-vs-throughput behavior as we increase the number of queries processed at a time. We see that as we increase the number of queries processed simultaneously, there

is an increase in both latency and throughput before the throughput stabilizes around 700 queries/second with about 30 queries. The throughput is relatively unaffected after this point.

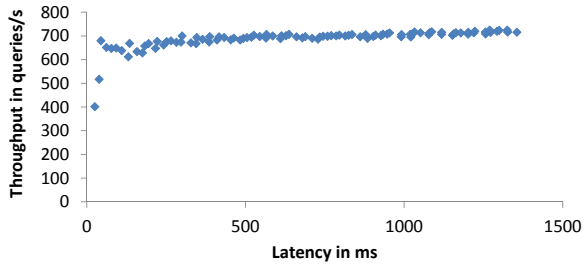

Figure 10: Latency Vs Throughput for processing queries in LSH. The size of the query set is varied from 10 to 1000 in steps of 10.

8.6 Streaming

As mentioned in Section 6, twitter is highly dynamic with about 400 million tweets added per day. We need to ensure that query performance does not drop below 1.5X of static queries, while still allowing for dynamic insertions. Our system achieves this.

Figure 11 shows the query performance of the system running on a single node. The overall capacity C of the node is 10.5 million tweets. The maximum size of the delta table structure (η) is 10% of the total capacity - around 1 million. The two lines in the graph represent query behavior under different amounts of data in the static tables. With about 50% of the overall capacity in the static structure, we achieve no performance degradation at all compared to fully static tables at capacity. With 90% of the capacity in the static structure, we encounter query runtimes going up to 1.3× that of 100% static tables in the worst case. Under average conditions, our streaming implementation will perform with no penalties.

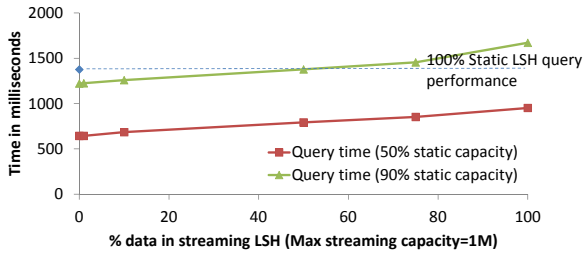

Figure 11: Streaming performance for 1000 queries with a constant amount of data in static table while data is added to the streaming tables. The dotted line represents the performance of 100% static LSH. Even in the worst case, the performance does not drop below 1.5X of static performance.

Insertion into the streaming LSH tables happens in chunks of 100k tweets. Each such insertion takes about 400 ms. We merge the streaming tables with the static tables when the streaming tables reach their capacity of 1 million tweets. This merge takes 15 seconds in the worst case (when static LSH is almost full). Hence, processing 1 million streaming tweets takes 19 seconds. Given the streaming rate of 400 million tweets/day and 4 nodes to handle the streaming data, the insert/merge processing takes a total of about 30 minutes in 24 hours. In other words, about 2% of the time is spent inserting and merging data.

9. DISCUSSION AND CONCLUSION

In this paper, we presented PLSH, a system to handle nearest neighbor queries on large amounts of text data based on an efficient parallel LSH implementation. PLSH is an in-memory, multithreaded distributed system capable of handling large amounts of streaming data while delivering very high query performance. We demonstrate its capabilities by running nearest neighbor queries on a billion tweet dataset on 100 nodes in 1–2.5 ms (per query), while streaming 100's of millions of tweets per day. We introduced several optimizations and a new variant that improves the time to build LSH tables by a factor of 3.7× and reduce query time by a factor of 8.3×. To the best of our knowledge, this makes PLSH the fastest LSH implementation currently available. We also introduced a performance model that is able to predict the performance of our implementation to within $15-25\%$, showing we are close to architectural limits and helping us pick the best PLSH parameters.

ACKNOWLEDGEMENTS

This work was supported by a grant from Intel, as a part of the Intel Science and Technology Center in Big Data (ISTC-BD).

10. REFERENCES

[1] E2LSH. http://www.mit.edu/~andoni/LSH/.
[2] LikeLike. http://code.google.com/p/likelike/.
[3] LSH-Hadoop. https://github.com/LanceNorskog/LSH-Hadoop.
[4] LSHKIT. http://lshkit.sourceforge.net.
[5] OptimalLSH. https://github.com/yahoo/Optimal-LSH.
[6] Twitter breaks 400 million tweet-per-day barrier, sees increasing mobile revenue. http://bit.ly/MmXObG.
[7] A. Andoni and P. Indyk. Efficient algorithms for substring near neighbor problem. In *Proceedings of SODA*, pages 1203–1212, 2006.
[8] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *CACM*, 58, 2008.
[9] N. Askitis and J. Zobel. Cache-conscious collision resolution in string hash tables. In *SPIRE*, pages 92–104, 2005.
[10] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *CIKM*, pages 2174–2178. ACM, 2012.
[11] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.
[12] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, Sept. 2001.
[13] M. Charikar. Similarity estimation techniques from rounding. In *Proceedings of STOC*, pages 380–388, 2002.
[14] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of SIGMOD*, 2005.
[15] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, 2007.
[16] A. Dasgupta, R. Kumar, and T. Sarlós. Fast locality-sensitive hashing. In *SIGKDD*, pages 1073–1081. ACM, 2011.
[17] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Inc., 1986.
[18] P. C.-M. K. A. P. Haghani. Lsh at large – distributed knn search in high dimensions. In *WebDB*, 2008.
[19] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.
[20] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. In *Proceedings of STOC*, 1998.
[21] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, et al. Sort vs. hash revisited: Fast join implementation on multi-core cpus. *PVLDB*, 2(2):1378–1389, 2009.
[22] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, 1986.
[23] Y. Li, J. M. Patel, and A. Terrell. Wham: A high-throughput sequence alignment method. *TODS*, 37(4):28:1–28:39, Dec. 2012.
[24] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proceedings of ACM SIGMOD*, 2006.
[25] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of WWW*, 2007.
[26] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2:185–197, 1991.
[27] J. Pan and D. Manocha. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *SIGSPATIAL*, 2011.
[28] S. Petrovic, M. Osborne, and V. Lavrenko. Streaming first story detection with application to twitter. In *NAACL*, volume 10, pages 181–189, 2010.
[29] A. Sadilek and H. Kautz. Modeling the impact of lifestyle on health at scale. In *Proceedings of ACM ICWSDM*, pages 637–646, 2013.
[30] N. Satish, C. Kim, J. Chhugani, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.
[31] M. Slaney, Y. Lifshits, and J. He. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9):2604–2623, 2012.
[32] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *Proceedings of SIGMOD*, 2005.