Efficient Distributed Locality Sensitive Hashing

Bahman Bahmani
Stanford Univesrity
Stanford, CA
bahman@stanford.edu

Ashish Goel
Stanford Univesrity
Stanford, CA
ashishg@stanford.edu

Rajendra Shinde
Stanford Univesrity
Stanford, CA
rbs@stanford.edu

ABSTRACT

(Key, Value) based distributed frameworks, such as MapReduce, Memcached, and Twitter Storm are gaining increasingly widespread use in applications that process large amounts of data. One important example application is large scale similarity search, for which Locality Sensitive Hashing (LSH) has emerged as the method of choice, specially when the data is high-dimensional. At its core, LSH is based on hashing the data points to a number of buckets such that similar points are more likely to map to the same buckets. To guarantee high search quality, the LSH scheme needs a rather large number of hash tables. This entails a large space requirement, and in the distributed setting, with each query requiring a network call per hash bucket look up, this also entails a big network load. The Entropy LSH scheme proposed by Panigrahy significantly reduces the number of required hash tables by looking up a number of query offsets in addition to the query itself. While this improves the LSH space requirement, it does not help with (and in fact worsens) the search network efficiency, as now each query offset requires a network call. In this paper, focusing on the Euclidian space under l_2 norm and building up on Entropy LSH, we propose the distributed Layered LSH scheme, and prove that it exponentially decreases the network cost, while maintaining a good load balance between different machines. Our experiments also verify that our scheme results in significant network traffic reductions which also bring about runtime improvements.

1. INTRODUCTION

Similarity search is the problem of retrieving data objects similar to a query object. It has become an important component of modern data-mining systems, with applications ranging from de-duplication of web documents, content-based audio, video, and image search [26, 29, 11], collaborative filtering [13], large scale genomic sequence alignment [9], natural language processing [33], pattern classification [12], and clustering [6].

In these applications, objects are usually represented by a high dimensional feature vector. Then, similarity search reduces to a variant of the Nearest Neighbor (NN) search problem, such as K-Nearest Neighbor (KNN) or Approximate Nearest Neighbor (ANN) search. A scheme to solve the NN search problem constructs an index which, given a query point, allows for quickly finding the data point closest to it. In addition to the query search procedure, the index construction also needs to be time and space efficient. Furthermore, since today's massive datasets are typically stored and processed in a distributed fashion, where network communication is one of the most important bottlenecks, these methods need to be network efficient, as otherwise, the network load would slow down the whole scheme.

There is a family of tree-based indexing methods for NN search, such as K-D trees [5], cover trees [7], navigating nets [25], R-trees [20], and SR-trees [24]. But, these techniques do not scale well with the data dimension, and it has been empirically observed [35] that with even modest dimensions, they are actually less efficient than linear scan of the data.

On the other hand, there is another family of ANN search methods based on the notion of Locality Sensitive Hashing (LSH) [23]. At its core, LSH is based on hashing the (data and query) points into a number of hash buckets such that similar points have higher chances of getting mapped to the same buckets. Then for each query, the nearest neighbor among the data points mapped to a same bucket as the query point is returned as the search result.

Unlike tree-based methods, LSH has been shown to scale well with the data dimension [23, 27]. However, the main drawback of conventional LSH based schemes is that to guarantee a good search quality, one needs a large number of hash tables. This entails a rather large space requirement for the index, and also in the distributed setting, a large network load, as each hash bucket look up requires a communication over the network. To mitigate the space efficiency issue, Panigrahy [32] proposed the Entropy LSH scheme, which significantly reduces the number of required hash tables, by looking up a number of query offsets in addition to the query itself. Even though this scheme improves the LSH space efficiency, it does not help with its network efficiency, as now each query offset lookup requires a network call. In fact, since the number of required offsets in Entropy LSH is larger than the number of required hash tables in conventional LSH, Entropy LSH amplifies the network inefficiency issue.

In this paper, focusing on the Euclidian space under l_2 norm and building up on the Entropy LSH scheme, we design the Layered LSH method for distributing the hash buckets over a set of machines which leads to an extremely network efficient distributed LSH scheme. We prove that, compared to a straightforward distributed implementation of LSH or Entropy LSH, our Layered LSH method results in an exponential improvement in the network load (from polynomial in n, the number of data points, to sub-logarithmic in n), while maintaining a good load balance between the different machines. Our experiments also verify that our scheme results in large network traffic improvements which in turn results in significant runtime speedups.

In the rest of this section, we first provide some background on the similarity search problem and the relevant methods, then discuss LSH in the distributed computation model, and finally present an overview of our scheme as well as our results.

1.1 Background

In this section, we briefly review the similarity search problem, the basic LSH and Entropy LSH approaches to solving it, the distributed computation framework and its instantiations such as MapReduce and Active DHT, and a straightforward implementation of LSH in the distributed setting as well as its major drawback.

Similarity Search: The similarity search problem is that of finding data objects similar to a query object. In many practical applications, the objects are represented by multidimensional feature vectors, and hence the problem reduces to finding objects closest to the query object under the feature space distance metric. This latter problem is usually denoted as the Nearest Neighbor (NN) search problem. There are a few variations of the NN search problem: K-Nearest Neighbor (KNN) search problem requires finding the K nearest data points to the query point, and the Approximate Nearest Neighbor (ANN) search problem requires finding a data point whose distance to the query point is almost equal to the distance of the closest data point. The goal in all these problems is to construct an index, which given the query point, allows for quickly finding the search result. The index construction and the query search both need to be space, time, and network efficient.

Basic LSH: Indyk and Motwani [23] proposed using a specific type of hash functions, namely Locality Sensitive Hash (LSH) functions, to solve the NN search problem over high dimensional large datasets. An LSH function maps the points in the feature space to a number of buckets in a way that similar points map to the same buckets with a high chance. Then, a similarity search query can be answered by first hashing the query point and then finding the close data points in the same bucket as the one the query is mapped to. To guarantee both a good search quality and a good search efficiency, one needs to use multiple LSH functions and combine their results. Then, although this approach yields a significant improvement in the running time over both the brute force linear scan and the space partitioning approaches [19, 5, 7, 25, 20, 24], unfortunately the required number of hash functions is usually large [9, 19], and since each hash table has the same size as the dataset, for large scale applications, this entails a very large space requirement for the index. Also, in the distributed setting, since each hash table lookup at query time corresponds to a network call, this entails a large network load which is also undesirable.

Entropy LSH: To mitigate the space inefficiency of LSH, Panigrahy [32] introduced the Entropy LSH scheme. This scheme uses the same hash functions and indexing method as the basic LSH scheme. However, it uses a different query time procedure: In addition to hashing the query point, it hashes a number of query offsets as well and also looks up the hash buckets that any of these offsets map to. The idea is that the close data points are very likely to be mapped to either the same bucket as the query point or to the same bucket as one of the query offsets. This significantly reduces the number of hash tables required to guarantee the search quality and efficiency. Hence, this scheme significantly improves the index space requirement compared to the basic LSH method. However, it unfortunately does not help with the query network efficiency, as each query offset requires a network call. Indeed, since one can see that [23, 32, 29] the number of query offsets required by Entropy LSH is larger than the number of hash tables required by basic LSH, the query network efficiency of Entropy LSH is even worse than that of the basic LSH.

In this paper, we focus on the network efficiency of LSH in distributed frameworks. Two main instantiations of such frameworks are the batched processing system MapReduce [16] (with its open source implementation Hadoop [1]), and the real-time processing systems denoted as Active Distributed Hash Tables (Active DHTs), such as Yahoo S4 [31] and Twitter Storm [2]. The common feature in all these systems is that they process data in the form of (Key, Value) pairs, distributed over a set of machines. This distributed (Key, Value) abstraction is all we need for both our scheme and analyses to apply. However, to make the later discussions more concrete, here we briefly overview the mentioned distributed systems.

MapReduce: MapReduce [16] is a simple computation model for batched distributed processing of large scale data, using a large number of commodity machines. By automatically handling the lower level issues such as fault tolerance, it provides a simple computational abstraction, where computations are done in three phases. The Map phase reads a collection of (Key, Value) pairs from an input source, and by invoking a user defined Mapper function on each input element independently and in parallel, emits zero or more (Key, Value) pairs associated with that input element. The Shuffle phase then groups together all the Mapper-emitted (Key, Value) pairs sharing the same Key, and outputs each distinct group to the next phase. The Reduce phase invokes a user-defined Reducer function on each distinct group, independently and in parallel, and emits zero or more values to associate with the group's Key. The emitted (Key, Value) pairs can then be written on the disk or be the input of the Map phase in a following iteration. The open source system Hadoop [1] is a widely used implementation of this computational framework.

Active DHT: A DHT (Distributed Hash Table) is a distributed (Key, Value) store which allows Lookups, Inserts, and Deletes on the basis of the Key. The term *Active* refers to the fact that an arbitrary User Defined Function (UDF) can be executed on a (Key, Value) pair in addition to Insert, Delete, and Lookup. Yahoo's S4 [31] and Twitter's Storm [2] are two examples of Active DHTs which are gaining widespread use. The Active DHT model is broad enough to act as a distributed stream processing system and as a continuous version of MapReduce [28]. All the (Key, Value) pairs in a node of the active DHT are usually stored in main memory. This requires that the distributed deployment has sufficient number of nodes, and will allow for fast real-time processing of data and queries.

In addition to the typical performance measures of total running time and total space, two other measures are very important for both MapReduce and Active DHTs:

1. total network traffic generated, that is the shuffle size for MapReduce and the number of network calls for Active DHT
2. the maximum number of values with the same key; a high value here can lead to the "curse of the last reducer" in MapReduce or to one compute node becoming a bottleneck in Active DHT.

A Simple Distributed LSH Implementation: We briefly discuss a simple implementation of LSH in the distributed framework. Each hash table associates a (Key, Value) pair to each data point, where the Key is the point's hash bucket, and the Value is the point itself. These (Key, Value) pairs are randomly distributed over the set of machines such that all the pairs with the same Key are on the same machine. This is done implicitly using a random hash function of the Key. For each query, first a number of (Key, Value) pairs corresponding to the query point are generated. The Value in all of these pairs is the query point itself. For basic LSH, per hash table, the Key is the hash bucket the query maps to, and for Entropy LSH, per query offset, the Key is the hash bucket the offset maps to. Then, each of these (Key, Value) pairs gets sent to and processed by the machine responsible for its Key. This machine contains all data points mapping to the same query or offset hash bucket. Then, it can perform a search within the data points which also map to the same Key and report the close points. This search can be done using the UDF in Active DHT or the Reducer in MapReduce.

In the above implementation, the amount of network communication per query is directly proportional to the number of hash buckets that need to be checked. However, as mentioned earlier, this number is large for both basic LSH and Entropy LSH. Hence, in large scale applications, where either there is a huge batch of queries or the queries arrive in real-time at very high rates, this will require a lot of communication, which not only depletes the valuable network resources in a shared environment, but also significantly slows down the query search process. In this paper, we propose an alternative way, called Layered LSH, to implement the Entropy LSH scheme in a distributed framework and prove that it exponentially reduces the network load compared to the above implementation, while maintaining a good load balance between different machines.

1.2 Overview of Our Scheme

At its core, Layered LSH is a carefully designed implementation of Entropy LSH in the distributed (Key, Value) model. The main idea is to distribute the hash buckets such that near points are likely to be on the same machine (hence network efficiency) while far points are likely to be on different machines (hence load balance).

This is achieved by rehashing the buckets to which the data points and the offsets of query points map to, via an additional layer of LSH, and then using the hashed buckets as Keys. More specifically, each data point is associated with a (Key, Value) pair where Key is the mapped value of LSH bucket containing the point, and Value is the point's hash bucket concatenated with the point itself. Also, each query point is associated with multiple (Key, Value) pairs where Value is the query itself and Keys are the mapped values of the buckets which need to be searched in order to answer this query.

Use of an LSH to rehash the buckets not only allows using the proximity of query offsets to bound the number of (Key, Value) pairs for each query (thus guaranteeing network efficiency), but also ensures that far points are unlikely to be hashed to the same machine (thus maintaining load balance).

1.3 Our Results

Here, we present a summary of our results in this paper:

1. We design a new scheme, called Layered LSH, to implement Entropy LSH in the distributed (Key, Value) model, and provide its pseudo-code for the two major distributed frameworks MapReduce and Active DHT.
2. We prove that Layered LSH incurs only $O(\sqrt{\log n})$ network cost per query. This is an exponential improvement over the $O(n^{\Theta(1)})$ query network cost of the simple distributed implementation of both Entropy LSH and basic LSH.
3. We prove that surprisingly the network efficiency of Layered LSH is independent of the search quality! This is in sharp contrast with both Entropy LSH and basic LSH in which increasing search quality directly increases the network cost. This offers a very large improvement in both network efficiency and hence overall run time in settings which require similarity search with high accuracy. We also present experiments which verify this observation on the MapReduce framework.
4. We prove that despite network efficiency (which requires collocating near points on the same machines), Layered LSH sends points which are only $\Omega(1)$ apart to different machines with high likelihood. This shows Layered LSH hits the right tradeoff between network efficiency and load balance across machines.
5. We present experimental results with Layered LSH on Hadoop, which show a factor 50 to 100 improvement in the "shuffle size" (i.e., the amount of data transmitted over the network) and 200 to 400 percent improvement on average in runtime, over a wide range of parameter values, on both synthetic and real world data sets.

The organization of this paper is as follows. In section 2, we study the Basic and Entropy LSH indexing methods. In section 3, we give the detailed description of Layered LSH, including its pseudocode for the MapReduce and Active DHT frameworks, and also provide the theoretical analysis of its network cost and load balance. We present experiments with Simple and Layered LSH on Hadoop in section 4, study the related work in section 5, and conclude in section 6.

2. PRELIMINARIES

In section 1.1, we provided the high-level background needed for this paper. Here, we present the necessary preliminaries in further detail. Specifically, we formally define the NN search problem, the notion of LSH functions, the basic LSH indexing, and Entropy LSH indexing.

NN Search: As mentioned in section 1.1, similarity search often reduces to an instantiation of the NN search problem. Here, we present the formal definition of this latter problem. Let T be the domain of all data and query objects, and ζ to be the distance metric between the objects. Given an approximation ratio $c > 1$, the c-ANN problem is that of constructing an index that given any query point $q \in T$, allows for quickly finding a data point $p \in T$ whose distance to q, as measured by ζ, is at most c times larger than the distance from q to its nearest data point. One can see that [22, 23] this problem can be solved by reducing it to the (c, r)-NN problem, in which the goal is to return a data point within distance cr of the query point q, given that there exists a data point within distance r of q.

Basic LSH: To solve the (c, r)-NN problem, Indyk and Motwani [23] introduced the following notion of LSH functions:

Definition 1. *For the space T with metric ζ, given distance threshold r, approximation ratio $c > 1$, and probabilities $p_1 > p_2$, a family of hash functions $\mathcal{H} = \{h : T \to U\}$ is said to be a (r, cr, p_1, p_2)-LSH family if for all $x, y \in T$,*

$$\begin{aligned} &\textit{if } \zeta(x,y) \le r \textit{ then } \mathbf{Pr}_{\mathcal{H}}[h(x) = h(y)] \ge p_1, \\ &\textit{if } \zeta(x,y) \ge cr \textit{ then } \mathbf{Pr}_{\mathcal{H}}[h(x) = h(y)] \le p_2. \end{aligned} \tag{2.1}$$

Hash functions drawn from $\mathcal{H}$ have the property that near points (with distance at most r) have a high likelihood (at least p_1) of being hashed to the same value, while far away points (with distance at least cr) are less likely (probability at most p_2) to be hashed to the same value; hence the name locality sensitive.

LSH families can be used to design an index for the (c, r)-NN problem as follows. First, for an integer k, let $\mathcal{H}' = \{H : T \to U^k\}$ be a family of hash functions in which any $H \in \mathcal{H}'$ is the concatenation of k functions in $\mathcal{H}$, i.e., $H = (h_1, h_2, \dots, h_k)$, where $h_i \in \mathcal{H}$ $(1 \le i \le k)$. Then, for an integer M, draw M hash functions from $\mathcal{H}'$, independently and uniformly at random, and use them to construct the index consisting of M hash tables on the data points. With this index, given a query q, the similarity search is done by first generating the set of all data points mapping to the same bucket as q in at least one hash table, and then finding the closest point to q among those data points. The idea is that a function drawn from $\mathcal{H}'$ has a very small chance (p_2^k) to map far away points to the same bucket (hence search efficiency), but since it also makes it less likely (p_1^k) for a near point to map to the same bucket, we use a number, M, of hash tables to guarantee retrieving the near points with a good chance (hence search quality).

To utilize this indexing scheme, one needs an LSH family $\mathcal{H}$ to start with. Such families are known for a variety of metric spaces, including the Hamming distance, the Earth Mover Distance, and the Jaccard measure [10]. Furthermore, Datar et al. [15] proposed LSH families for l_p norms, with $0 \le p \le 2$, using p-stable distributions. For any $W > 0$, they consider a family of hash functions $\mathcal{H}_W : \{h_{\mathbf{a},b} : \mathbb{R}^d \to \mathbb{Z}\}$ such that

$$h_{\mathbf{a},b}(v) = \lfloor \frac{\mathbf{a} \cdot v + b}{W} \rfloor$$

where $\mathbf{a} \in \mathbb{R}^d$ is a d-dimensional vector each of whose entries is chosen independently from a p-stable distribution, and $b \in \mathbb{R}$ is chosen uniformly from $[0, W]$. Further improvements have been obtained in various special settings [4]. In this paper, we will focus on the most widely used p-stable distribution, i.e., the 2-stable, Gaussian distribution. For this case, Indyk and Motwani [23] proved the following theorem:

Theorem 2. *With n data points, choosing $k = O(\log n)$ and $M = O(n^{1/c})$, the LSH indexing scheme above solves the (c, r)-NN problem with constant probability.*

Entropy LSH: As explained in section 1.1, the main drawback of the basic LSH indexing scheme is the large number (in practice, up to hundreds [9, 19]) of hash tables it needs to get a good quality. This entails a large index space requirement, and in the distributed setting, also a big network load. To mitigate the space inefficiency, Panigrahy [32] introduced the Entropy LSH scheme. This scheme uses the same indexing as in the basic LSH scheme, but a different query search procedure. The idea here is that for each hash function $H \in \mathcal{H}'$, the data points close to the query point q are highly likely to hash either to the same value as $H(q)$ or to a value very close to that. Hence, it makes sense to also consider as candidates the points mapping to close hash values. To do so, in this scheme, in addition to q, several "offsets" $q + \delta_i$ $(1 \le i \le L)$, chosen randomly from the surface of $B(q, r)$, the sphere of radius r centered at q, are also hashed and the data points in their hash buckets are also considered as search result candidates. It is conceivable that this may reduce the number of required hash tables, and in fact, Panigrahy [32] shows that with this scheme one can use as few as $\tilde{O}(1)$ hash tables. The instantiation of his result for the l_2 norm is as follows:

Theorem 3. *For n data points, choosing $k \ge \frac{\log n}{\log(1/p_2)}$ (with p_2 as in Definition 1) and $L = O(n^{2/c})$, as few as $\tilde{O}(1)$ hash tables suffice to solve the (c, r)-NN problem.*

Hence, this scheme in fact significantly reduces the number of required hash tables (from $O(n^{1/c})$ for basic LSH to $\tilde{O}(1)$), and hence the space efficiency of LSH. However, in the distributed setting, it does not help with reducing the network load of LSH queries. Actually, since for the basic LSH, one needs to look up $M = O(n^{1/c})$ buckets but with this scheme, one needs to look up $L = O(n^{2/c})$ offsets, it makes the network inefficiency issue even more severe.

3. DISTRIBUTED LSH

In this section, we will present the Layered LSH scheme and theoretically analyze it. We will focus on the d-dimensional Euclidian space under l_2 norm. As notation, we will let S to be a set of n data points available a-priori, and Q to be the set of query points, either given as a batch (in case of MapReduce) or arriving in real-time (in case of Active DHT). Parameters k, L, W and LSH families $\mathcal{H} = \mathcal{H}_W$ and $\mathcal{H}' = \mathcal{H}'_W$ will be as defined in section 2. Since multiple hash tables can be obviously implemented in parallel, for the sake of clarity we will focus on a single hash table and use a randomly chosen hash function $H \in \mathcal{H}'$ as our LSH function throughout the section.

In (Key, Value) based distributed systems, a hash function from the domain of all Keys to the domain of available machines is implicitly used to determine the machine responsible for each (Key, Value) pair. In this section, for the sake of clarity, we will assume this mapping to be simply identity. That is, the machine responsible for a (Key, Value) data element is simply the machine with id equal to Key.

At the core, Layered LSH is a carefully distributed implementation of Entropy LSH. Hence before presenting it, first we further detail the simple distributed implementation of Entropy LSH, described in section 1.1, and explain its major drawback. For any data point $p \in S$ a (Key, Value) pair $(H(p), p)$ is generated and sent to machine $H(p)$. For each query point q, after generating the offsets $q+\delta_i$ $(1 \leq i \leq L)$, for each offset $q + \delta_i$ a (Key, Value) pair $(H(q + \delta_i), q)$ is generated and sent to machine $H(q+\delta_i)$. Hence, a machine x will have all the data points $p \in S$ with $H(p) = x$ as well as all query points $q \in Q$ such that $H(q + \delta_i) = x$ for some $1 \leq i \leq L$. Then, for any received query point q, this machine retrieves all data points p with $H(p) = x$ which are within distance cr of q, if any such data points exist. This is done via a UDF in Active DHT or the Reducer in MapReduce, as presented in Figure 3.1 for the sake of concreteness of exposition.

In this implementation, the network load due to data points is not very significant. Not only just one (Key, Value) pair per data point is transmitted over the network, but also in many real-time applications, data indexing is done offline where efficiency and speed are not as critical. However, the amount of data transmitted per query in this implementation is $O(Ld)$: L (Key, Value) pairs, one per offset, each with the d-dimensional point q as Value. Both L and d are large in many practical applications with high-dimensional data (e.g., L can be in the hundreds, and d in the tens or hundreds). Hence, this implementation needs a lot of network communication per query, and with a large batch of queries or with queries arriving in real-time at very high rates, this will not only put a lot of strain on the valuable and usually shared network resources but also significantly slow down the search process.

Therefore, a distributed LSH scheme with significantly better query network efficiency is needed. This is where Layered LSH comes into the picture.

3.1 Layered LSH

In this subsection, we present the Layered LSH scheme. The main idea is to use another layer of locality sensitive hashing to distribute the data and query points over the machines. More specifically, given a parameter value $D > 0$, we sample an LSH function $G : \mathbb{R}^k \to \mathbb{Z}$ such that:

$$G(v) = \lfloor \frac{\alpha \cdot v + \beta}{D} \rfloor \tag{3.1}$$

where $\alpha \in \mathbb{R}^k$ is a k-dimensional vector whose individual entries are chosen from the standard Gaussian $\mathcal{N}(0, 1)$ distribution, and $\beta \in \mathbb{R}$ is chosen uniformly from $[0, D]$.

Then, denoting $G(H(\cdot))$ by $GH(\cdot)$, for each data point $p \in S$, we generate a (Key, Value) pair $(GH(p), < H(p), p >)$, which gets sent to machine $GH(p)$. By breaking down the Value part to its two pieces, $H(p)$ and p, this machine will then add p to the bucket $H(p)$. This can be done by the Reducer in MapReduce, and by a UDF in Active DHT. Similarly, for each query point $q \in Q$, after generating the offsets $q + \delta_i$ $(1 \leq i \leq L)$, for each unique value x in the set

$$\{GH(q + \delta_i) | \, 1 \leq i \leq L\} \tag{3.2}$$

we generate a (Key, Value) pair (x, q) which gets sent to machine x. Then, machine x will have all the data points p such that $GH(p) = x$ as well as the queries $q \in Q$ one of whose offsets gets mapped to x by $GH(\cdot)$. Specifically, if for the offset $q+\delta_i$, we have $GH(q+\delta_i) = x$, all the data points p that $H(p) = H(q+\delta_i)$ are also located on machine x. Then, this machine regenerates the offsets $q+\delta_i$ $(1 \leq i \leq L)$, finds their hash buckets $H(q + \delta_i)$, and for any of these buckets such that $GH(q + \delta_i) = x$, it performs a NN search among the data points in that bucket. Note that since q is sent to this machine, there exists at least one such bucket. Also note that, the offset regeneration, hash, and bucket search can all be done by either a UDF in Active DHT or the Reducer in MapReducer. To make the exposition more concrete, we have presented the pseudo code for both the MapReduce and Active DHT implementations of this scheme in Figure 3.2.

At an intuitive level, the main idea in Layered LSH is that since G is an LSH, and also for any query point q, we have $H(q + \delta_i) \simeq H(q)$ for all offsets $q + \delta_i$ $(1 \leq i \leq L)$, the set in equation 3.2 has a very small cardinality, which in turn implies a small amount of network communication per query. On the other hand, since G and H are both LSH functions, if two data points p, p' are far apart, $GH(p)$ and $GH(p')$ are highly likely to be different. This means that, while locating the nearby points on the same machines, Layered LSH partitions the faraway data points on different machines, which in turn ensures a good load balance across the machines. Note that this is critical, as without a good load balance, the point in distributing the implementation would be lost.

In the next section, we present the formal analysis of this scheme, and prove that compared to the simple implementation, it provides an exponential improvement in the network

Algorithm 1 MapReduce Implementation

```
**Map:**
**Input:** Data set $S$, query set $Q$
Choose $H$ from $\mathcal{H}'_W$ uniformly at random, but consistently across Mappers
**for** each data point $p \in S$ **do**
  Emit $(H(p), p)$
**end for**
**for** each query point $q \in Q$ **do**
  **for** $1 \leq i \leq L$ **do**
    Choose the offset $q + \delta_i$ from the surface of $B(q, r)$
    Emit $(H(q + \delta_i), q)$
  **end for**
**end for**

**Reduce:**
**Input**: For a hash bucket $x$, all data points $p \in S$ with $H(p) = x$, and all query points $q \in Q$ one of whose offsets hashes to $x$.
**for** each query point $q$ among the input points **do**
  **for** each data point $p$ among the input points **do**
    **if** $p$ is within distance $cr$ of $q$ **then**
      Emit $(q, p)$
    **end if**
  **end for**
**end for**
```

Algorithm 2 Active DHT Implementation

```
**Preprocessing:**
**Input:** Data set $S$
**for** each data point $p \in S$ **do**
  Compute the hash bucket $y = H(p)$
  Send the pair $(y, p)$ to machine with id $y$
  At machine $y$ add $p$ to the in-memory bucket $y$
**end for**

**Query Time:**
**Input:** Query point $q \in Q$ arriving in real-time
**for** $1 \leq i \leq L$ **do**
  Generate the offset $q + \delta_i$
  Compute the hash bucket $x = H(q + \delta_i)$
  Send the pair $(x, q)$ to machine with id $x$
  At machine $x$, run SearchUDF$(x, q)$
**end for**

**SearchUDF**$(x, q)$**:**
**for** each data point $p$ with $H(p) = x$ **do**
  **if** $p$ is within distance $cr$ of $q$ **then**
    Emit $(q, p)$
  **end if**
**end for**
```

Figure 3.1: Simple Distributed LSH

traffic, while maintaining a good load balance across the machines.

3.2 Analysis

In this section, we analyze the Layered LSH scheme presented in the previous section. We first fix some notation. As mentioned earlier in the paper, we are interested in the (c, r)-NN problem. Without loss of generality and to simplify the notation, in this section we assume $r = 1/c$. This can be achieved by a simple scaling. The LSH function $H \in \mathcal{H}'_W$ that we use is $H = (H_1, \ldots, H_k)$, where k is chosen as in Theorem 3 and for each $1 \leq i \leq k$:

$$H_i(v) = \lfloor \frac{a_i \cdot v + b_i}{W} \rfloor$$

where a_i is a d-dimensional vector each of whose entries is chosen from the standard Gaussian $\mathcal{N}(0, 1)$ distribution, and $b_i \in \mathbb{R}$ is chosen uniformly from $[0, W]$. We will also let $\Gamma : \mathbb{R}^d \to \mathbb{R}^k$ be $\Gamma = (\Gamma_1, \ldots, \Gamma_k)$, where for $1 \leq i \leq k$:

$$\Gamma_i(v) = \frac{a_i \cdot v + b_i}{W}$$

hence, $H_i(\cdot) = \lfloor \Gamma_i(\cdot) \rfloor$. We will use the following small lemma in our analysis:

LEMMA 4. *For any two vectors $u, v \in \mathbb{R}^d$, we have:*

$$||\Gamma(u) - \Gamma(v)|| - \sqrt{k} \leq ||H(u) - H(v)|| \leq ||\Gamma(u) - \Gamma(v)|| + \sqrt{k}$$

PROOF. Denoting $R_i = \Gamma_i - H_i$ $(1 \leq i \leq k)$ and $R = (R_1, \ldots, R_k)$, we have $0 \leq R_i(u), R_i(v) \leq 1$ $(1 \leq i \leq k)$, and hence $||R(u) - R(v)|| \leq \sqrt{k}$. Also, by definition $H = \Gamma - R$, and hence $H(u) - H(v) = (\Gamma(u) - \Gamma(v)) + (R(v) - R(u))$. Then, the result follows from triangle inequality. □

Our analysis also uses two well-known facts. The first is the sharp concentration of χ^2-distributed random variables, which is also used in the proof of the Johnson-Lindenstrauss lemma [23, 14]:

FACT 5. *If $\omega \in \mathbb{R}^m$ is a random m-dimensional vector each of whose entries is chosen from the standard Gaussian $\mathcal{N}(0, 1)$ distribution, and $m = \Omega(\frac{\log n}{\epsilon^2})$, then with probability at least $1 - \frac{1}{n^{\Theta(1)}}$, we have*

$$(1 - \epsilon)\sqrt{m} \leq ||\omega|| \leq (1 + \epsilon)\sqrt{m}$$

The second fact is the 2-stability property of Gaussian distribution:

FACT 6. *If θ is a vector each of whose entries is chosen from the standard Gaussian $\mathcal{N}(0, 1)$ distribution, then for any vector v of the same dimension, the random variable $\theta \cdot v$ has Gaussian $\mathcal{N}(0, ||v||)$ distribution.*

The plan for the analysis is as follows. We will first analyze (in theorem 8) the network traffic of Layered LSH and derive a formula for it based on D, the bin size of LSH function G. We will see that as expected, increasing D reduces the

Algorithm 3 MapReduce Implementation

```
Map:
Input: Data set $S$, query set $Q$
Choose hash functions $H, G$ randomly but consistently across mappers
for each data point $p \in S$ do
    Emit $(GH(p), < H(p), p >)$
end for
for each query point $q \in Q$ do
    for $1 \le i \le L$ do
        Generate the offset $q + \delta_i$
        Emit $(GH(q + \delta_i), q)$
    end for
end for

Reduce:
Input: For a hash bucket $x$, all pairs $< H(p), p >$ for data points $p \in S$ with $GH(p) = x$, and all query points $q \in Q$ one of whose offsets is mapped to $x$ by $GH$.
for each data point $p$ among the input points do
    Add $p$ to bucket $H(p)$
end for
for each query point $q$ among the input points do
    for $1 \le i \le L$ do
        Generate the offset $q + \delta_i$, and find $H(q + \delta_i)$
        if $GH(q+\delta_i) = x, H(q+\delta_i) \neq H(q+\delta_j)$ $(\forall j < i)$ then
            for each data point $p$ in bucket $H(q + \delta_i)$ do
                if ($p$ is within distance $cr$ of $q$) then
                    Emit $(q, p)$
                end if
            end for
        end if
    end for
end for
```

Algorithm 4 Active DHT Implementation

```
Preprocessing:
Input: Data set $S$
for each data point $p \in S$ do
    Compute the hash bucket $H(p)$ and machine id $y = GH(p)$
    Send the pair $(y, < H(p), p >)$ to machine with id $y$
    At machine $y$, add $p$ to the in-memory bucket $H(p)$
end for

Query Time:
Input: Query point $q \in Q$ arriving in real-time
for $1 \le i \le L$ do
    Generate the offset $q + \delta_i$, compute $x = GH(q + \delta_i)$
    if $GH(q + \delta_j) \neq x$ $(\forall j < i)$ then
        Send the pair $(x, q)$ to machine with id $x$
        At machine $x$, run SearchUDF$(x, q)$
    end if
end for

SearchUDF$(x, q)$:
for $1 \le i \le L$ do
    Generate offset $q+\delta_i$, compute $H(q+\delta_i), GH(q+\delta_i)$
    if $GH(q + \delta_i) = x, H(q + \delta_i) \neq H(q + \delta_j)$ $(\forall j < i)$ then
        for each data point $p$ in bucket $H(q + \delta_i)$ do
            if $p$ is within distance $cr$ from $q$ then
                Emit $(q, p)$
            end if
        end for
    end if
end for
```

Figure 3.2: Layered LSH

network traffic, and our formula will show the exact relation between the two. We will next analyze (in theorem 11) the load balance of Layered LSH and derive a formula for it, again based on D. Intuitively speaking, a large value of D tends to put all points on one or few machines, which is undesirable from the load balance perspective. Our analysis will formulate this dependence and show its exact form. These two results together will then show the exact trade-off governing the choice of D, which we will use to prove (in Corollary 12) that with an appropriate choice of D, Layered LSH achieves both network efficiency and load balance. Before proceeding to the analysis, we give a definition:

Definition 7. *Having chosen LSH functions G, H, for a query point $q \in Q$, with offsets $q + \delta_i$ $(1 \le i \le L)$, define*

$$f_q = |\{GH(q + \delta_i) | 1 \le i \le L\}|$$

to be the number of (Key, Value) pairs sent over the network for query q.

Since q is d-dimensional, the network load due to query q is $O(df_q)$. Hence, to analyze the network efficiency of Layered LSH, it suffices to analyze f_q. This is done in the following theorem:

Theorem 8. *For any query point q, with high probability, that is probability at least $1 - \frac{1}{n^{\Theta(1)}}$, we have:*

$$f_q = O(\frac{k}{D})$$

Proof. Since for any offset $q + \delta_i$, the value $GH(q + \delta_i)$ is an integer, we have:

$$f_q \le \max_{1 \le i,j \le L} \{GH(q + \delta_i) - GH(q + \delta_j)\} \tag{3.3}$$

For any vector v, we have:

$$\frac{\alpha.v + \beta}{D} - 1 \le G(v) \le \frac{\alpha.v + \beta}{D}$$

hence for any $1 \le i, j \le L$:

$$GH(q + \delta_i) - GH(q + \delta_j) \le \frac{\alpha \cdot (H(q + \delta_i) - H(q + \delta_j))}{D} + 1$$

Thus, from equation 3.3, we get:

$$f_q \leq \frac{1}{D} \max_{1\leq i,j\leq L} \{\alpha \cdot (H(q+\delta_i) - H(q+\delta_j))\} + 1$$

From Cauchy-Schwartz inequality for inner products, we have for any $1 \leq i, j \leq L$:

$$\alpha \cdot (H(q+\delta_i) - H(q+\delta_j)) \leq ||\alpha|| \cdot ||H(q+\delta_i) - H(q+\delta_j)||$$

Hence, we get:

$$f_q \leq \frac{||\alpha||}{D} \cdot \max_{1\leq i,j\leq L} \{||H(q+\delta_i) - H(q+\delta_j)||\} + 1 \quad (3.4)$$

For any $1 \leq i, j \leq L$, we know from lemma 4:

$$||H(q+\delta_i) - H(q+\delta_j)|| \leq ||\Gamma(q+\delta_i) - \Gamma(q+\delta_j)|| + \sqrt{k} \quad (3.5)$$

Furthermore for any $1 \leq t \leq k$, since

$$\Gamma_t(q+\delta_i) - \Gamma_t(q+\delta_j) = \frac{a_t \cdot (\delta_i - \delta_j)}{W}$$

we know, using Fact 6, that $\Gamma_t(q+\delta_i) - \Gamma_t(q+\delta_j)$ is distributed as Gaussian $\mathcal{N}(0, \frac{||\delta_i - \delta_j||}{W})$. Now, recall from theorem 3 that for our LSH function, $k \geq \frac{\log n}{\log(1/p_2)}$. Hence, there is a constant $\epsilon = \epsilon(p_2) < 1$ for which we have, using Fact 5:

$$||\Gamma(q+\delta_i) - \Gamma(q+\delta_j)|| \leq (1+\epsilon)\sqrt{k}\frac{||\delta_i - \delta_j||}{W}$$

with probability at least $1 - 1/n^{\Theta(1)}$. Since, as explained in section 2, all offsets are chosen from the surface of the sphere $B(q, 1/c)$ of radius $1/c$ centered at q, we have: $||\delta_i - \delta_j|| \leq 2/c$. Hence overall, for any $1 \leq i, j \leq k$:

$$||\Gamma(q+\delta_i) - \Gamma(q+\delta_j)|| \leq 2(1+\epsilon)\frac{\sqrt{k}}{cW} \leq 4\frac{\sqrt{k}}{cW}$$

with high probability. Then, since there are only L^2 different choices of i, j, and L is only polynomially large in n, we get:

$$\max_{1\leq i,j\leq L} \{||\Gamma(q+\delta_i) - \Gamma(q+\delta_j)||\} \leq 4\frac{\sqrt{k}}{cW}$$

with high probability. Then, using equation 3.5, we get with high probability:

$$\max_{1\leq i,j\leq L} \{||H(q+\delta_i) - H(q+\delta_j)||\} \leq (1 + \frac{4}{cW})\sqrt{k} \quad (3.6)$$

Furthermore, since each entry of $\alpha \in \mathbb{R}^k$ is distributed as $\mathcal{N}(0,1)$, another application of Fact 5 gives (again with $\epsilon = \epsilon(p_2)$):

$$||\alpha|| \leq (1+\epsilon)\sqrt{k} \leq 2\sqrt{k} \quad (3.7)$$

with high probability. Then, equations 3.4, 3.6, and 3.7 together give:

$$f_q \leq 2(1 + \frac{4}{cW})\frac{k}{D} + 1$$

which finishes the proof. □

REMARK 9. *A surprising property of Layered LSH demonstrated by theorem 8 is that the network load is independent of the number of query offsets, L. Note that with Entropy LSH, to increase the search quality one needs to increase the number of offsets, which will then directly increase the network load. Similarly, with basic LSH, to increase the search quality one needs to increase the number of hash tables, which again directly increases the network load. However, with Layered LSH the network efficiency is achieved independently of the level of search quality. Hence, search quality can be increased without any effect on the network load!*

Next, we proceed to analyzing the load balance of Layered LSH. First, recalling the classic definition of error function:

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-\tau^2} d\tau$$

we define the function $P(\cdot)$:

$$P(z) = \text{erf(z)} - \frac{1}{\sqrt{\pi}z}(1 - e^{-z^2}) \quad (3.8)$$

and prove the following lemma:

LEMMA 10. *For any two points $u, v \in \mathbb{R}^k$ with $||u - v|| = \lambda$, we have:*

$$Pr[G(u) = G(v)] = P(\frac{D}{\sqrt{2}\lambda})$$

PROOF. Since β is uniformly distributed over $[0, D]$, we have:

$$Pr[G(u) = G(v) | \, \alpha \cdot (u - v) = l] = \max\left\{0, 1 - \frac{|l|}{D}\right\}$$

Then, since by Fact 6, $\alpha \cdot (u - v)$ is distributed as Gaussian $\mathcal{N}(0, \lambda)$, we have:

$$\begin{aligned} Pr[G(u) = G(v)] &= \int_{-D}^{D} (1 - \frac{|l|}{D})\frac{1}{\sqrt{2\pi}\lambda}e^{-\frac{l^2}{2\lambda^2}} dl \\ &= 2\int_0^D (1 - \frac{l}{D})\frac{1}{\sqrt{2\pi}\lambda}e^{-\frac{l^2}{2\lambda^2}} dl \\ &= \int_0^D \frac{\sqrt{2}}{\sqrt{\pi}\lambda}e^{-\frac{l^2}{2\lambda^2}} dl - \int_0^D \frac{l\sqrt{2}}{D\lambda\sqrt{\pi}}e^{-\frac{l^2}{2\lambda^2}} dl \\ &= \text{erf}(\frac{D}{\sqrt{2}\lambda}) - \sqrt{\frac{2}{\pi}}\frac{\lambda}{D}(1 - e^{-\frac{D^2}{2\lambda^2}}) \\ &= P(\frac{D}{\sqrt{2}\lambda}) \end{aligned}$$

□

One can easily see that $P(\cdot)$ is a monotonically increasing function, and for any $0 < \xi < 1$ there exists a number $z = z_\xi$ such that $P(z_\xi) = \xi$. Using this notation and the previous lemma, we prove the following theorem:

THEOREM 11. *For any constant $0 < \xi < 1$, there is a λ_ξ such that*

$$\lambda_\xi / W = O(1 + \frac{D}{\sqrt{k}})$$

and for any two points u, v with $||u - v|| \geq \lambda_\xi$, we have:

$$Pr[GH(u) = GH(v)] \leq \xi + o(1)$$

where $o(1)$ is polynomially small in n.

Proof. Let $u, v \in \mathbb{R}^d$ be two points and denote $||u-v|| = \lambda$. Then by lemma 4, we have:

$$||H(u) - H(v)|| \geq ||\Gamma(u) - \Gamma(v)|| - \sqrt{k}$$

As in the proof of theorem 8, one can see, using $k \geq \frac{\log n}{\log(1/p_2)}$ (from theorem 3) and Fact 5, that there exists an $\epsilon = \epsilon(p_2) = \Theta(1)$ such that with probability at least $1 - \frac{1}{n^{\Theta(1)}}$ we have:

$$||\Gamma(u) - \Gamma(v)|| \geq (1-\epsilon)\frac{\lambda\sqrt{k}}{W}$$

Hence, with probability at least $1 - \frac{1}{n^{\Theta(1)}}$, we have:

$$||H(u) - H(v)|| \geq \lambda' = ((1-\epsilon)\frac{\lambda}{W} - 1)\sqrt{k}$$

Now, letting:

$$\lambda_\xi = \frac{1}{1-\epsilon}(1 + \frac{D}{z_\xi\sqrt{2k}})W$$

we have if $\lambda \geq \lambda_\xi$ then $\lambda' \geq \frac{D}{\sqrt{2}z_\xi}$, and hence by lemma 10:

$$Pr[GH(u) = GH(v) | \, ||H(u) - H(v)|| \geq \lambda'] \leq P(\frac{D}{\sqrt{2}\lambda'}) \leq \xi$$

which finishes the proof by recalling $Pr[||H(u) - H(v)|| < \lambda'] = o(1)$. $\square$

Theorems 8, 11 show the tradeoff governing the choice of parameter D. Increasing D reduces network traffic at the cost of more skewed load distribution. We need to choose D such that the load is balanced yet the network traffic is low. Theorem 11 shows that choosing $D = o(\sqrt{k})$ does not asymptotically help with the distance threshold at which points become likely to be sent to different machines. On the other hand, theorem 11 also shows that choosing $D = \omega(\sqrt{k})$ is undesirable, as it unnecessarily skews the load distribution. To observe this more clearly, recall that intuitively speaking, the goal in Layered LSH is that if two data point p_1, p_2 hash to the same values as two of the offsets $q+\delta_i, q+\delta_j$ (for some $1 \leq i, j \leq L$) of a query point q (i.e., $H(p_1) = H(q+\delta_i)$ and $H(p_2) = H(q+\delta_j)$), then p_1, p_2 are likely to be sent to the same machine. Since H has a bin size of W, such pair of points p_1, p_2 most likely have distance $O(W)$. Hence, D should be only large enough to make points which are $O(W)$ away likely to be sent to the same machine. Theorem 11 shows that to do so, we need to choose D such that:

$$O(1 + \frac{D}{\sqrt{k}}) = O(1)$$

that is $D = O(\sqrt{k})$. Then, by theorem 8, to minimize the network traffic, we choose $D = \Theta(\sqrt{k})$, and get $f_q = O(\sqrt{k}) = O(\sqrt{\log n})$. This is summarized in the following corollary:

Corollary 12. *Choosing $D = \Theta(\sqrt{k})$, Layered LSH guarantees that the number of (Key, Value) pairs sent over the network per query is $O(\sqrt{\log n})$ with high probability, and yet points which are $\Omega(W)$ away get sent to different machines with constant probability.*

Remark 13. *Corollary 12 shows that, compared to the simple distributed implementation of Entropy LSH and basic LSH, Layered LSH exponentially improves the network load, from $O(n^{\Theta(1)})$ to $O(\sqrt{\log n})$, while maintaining the load balance across the different machines.*

4. EXPERIMENTS

In this section we present an experimental comparison of Simple and Layered LSH via the MapReduce framework which show a factor 50 to 100 reduction in the network cost (shuffle size) for Layered LSH, and on average, a 200 to 400% improvement in the "wall-clock" run time.

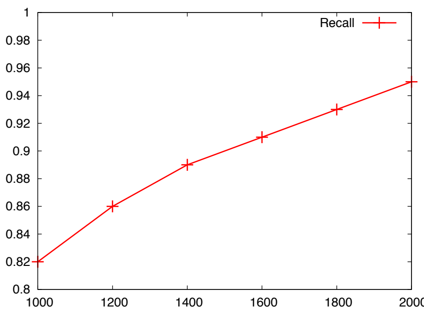
4.1 Datasets

We use two data sets, an artificially generated "Random" data set and an "Image" data set obtained from Tiny Images [3], as described below.

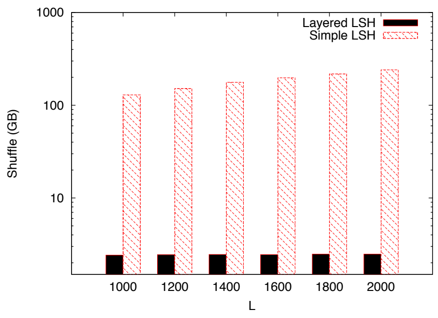
- **Random:** For the d-dimensional Euclidean space $\mathbb{R}^d$, let $N^d(\mathbf{0}, r)$ denote the normal distribution around the origin, $\mathbf{0}$, where the i-th coordinate of a randomly chosen point has the distribution $\mathcal{N}(0, r/\sqrt{d})$, $\forall i \in 1 \dots d$. We create a data set by sampling 1 million (1M) data points from $N^d(\mathbf{0}, 1)$ with $d = 100$ and choose hundred thousand (100K) query points by randomly selecting a data point and adding a small perturbation drawn from $N^d(\mathbf{0}, r)$ to it, where $r = 0.3$. We present the results on solving the (c, r)-NN problem on this data set with $c = 2$. The choice of the distribution as well as parameters (c, r) is such that for each query point, the expected distance to its nearest neighbor in the data set is r and that with high probability only the data point to which it was chosen to be near to, by design, will be at a distance at most cr from it. The data set and query set were chosen large enough to ensure that a non trivial shuffle component in the MapReduce implementation of Entropy LSH.
- **Image [3]:** The Tiny Image Data set consists of almost 80M "tiny" images of size 32×32 pixels [3]. We extract a 64-dimensional color histogram from each image in this data set using the *extractcolorhistogram* tool in the FIRE image search engine, as described in [29, 17] and normalize it to unit norm in the preprocessing step. Next, we randomly sample 1M data set and 200K query set points ensuring no overlap among data and query sets. Since $d = 64$, we chose the query set to be twice as large as the Random data set in order to ensure approximately the same network cost per query offset for the Entropy LSH method. Since the average distance of a query point to its nearest neighbor in this data set was 0.08 (with standard deviation 0.07), we present the results of solving the (c, R)-NN problem on this data set with parameters $(r = 0.08, c = 2)$ and $(r = 0.16, c = 2)$, noting that we obtain similar results for many other parameter choices.

4.2 Metrics

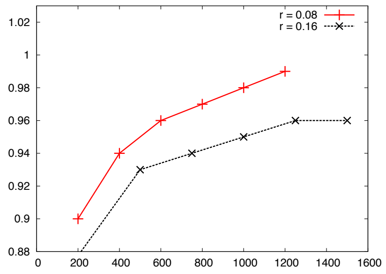
We compare the overall network efficiency of the Simple and Layered LSH using the shuffle size i.e., total amount of data shuffled over the network in the Shuffle phase ("Reduce shuffle bytes" in Hadoop [1]) and also compare the "wall-clock" running time of the two schemes.



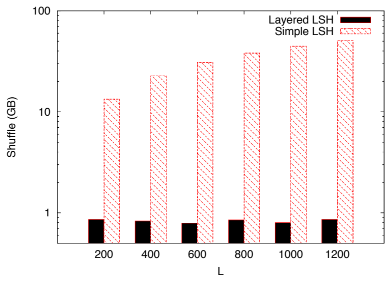
(a) Random: recall



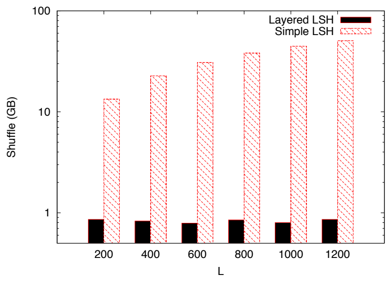
(b) Random: shuffle size



(c) Image: recall



(d) Image: shuffle size, $r = 0.08$



(e) Image: shuffle size, $r = 0.16$

Figure 4.1: Variation in recall and shuffle size (log scale) with L for the Random (Figures (a), (b)) and the Image (Figures (c), (d), (e)) data sets. Layered LSH yields a factor 100 improvement for the Random data set and on average a factor 50 improvement on the Image data set.

As described in Section 3, even though the Simple and Layered LSH have different distributed implementations, they output the same result. Consequently, evaluating the accuracy of the results (i.e. recall) is not an important aspect of this paper, since the effectiveness of LSH has been verified by many earlier studies [15, 19, 29]. Still, for completeness, (and to make sure our Layered LSH implementation does not have implementation errors,) we measure the accuracy of search results by computing the recall, i.e. the fraction of query points, having at least a single data point within a distance cr, returned in the output.

4.3 Implementation Details

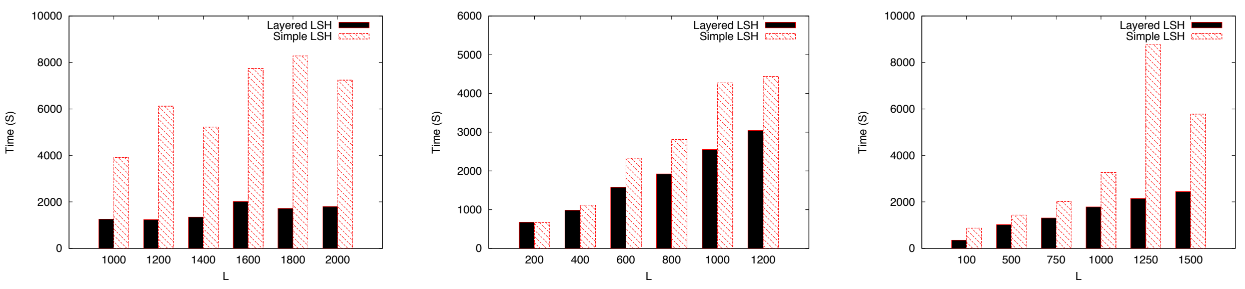
We perform experiments on a small cluster of 16 compute nodes using Hadoop version 0.20.2 with 512MB heap size for individual JVMs. Consistency in the choice of hash functions H, G for Simple and Layered LSH (figures 3.1, 3.2) across mappers is ensured by setting the seed of the random number generator(RNG) appropriately. Consistency in choice of offsets for each query point across mappers (and reducers in Layered LSH) is ensured by using a RNG with the query point itself as seed. We choose the LSH parameters ($W = 0.5, k = 10$) for the Random data set and ($W = 0.3, k = 16$) for the Image data set as per calculations in the Entropy LSH work and previous work on image data sets [32, 29]. We choose D via a binary search to minimize the "wall-clock" run time of Layered LSH and and report performance variation with increasing values of L, the number of query offsets. We note that a systematic understanding of the choice of parameters for Layered LSH (as well as Basic LSH) is an important research direction.

4.4 Results

Next we present performance variation experiments for Simple and Layered LSH by scaling L, the number of query offsets. In addition to this, we also study the sensitivity of performance with variation in D, the parameter of Layered LSH.

Scaling L:
Figure 4.1 describes the results of scaling L on the shuffle size and recall for both the data sets, where the shuffle sizes are plotted in a logarithmic scale. We observe that although increasing L improves the recall, it also results in a linear increase in the shuffle size for Simple LSH since a (Key, Value) pair is emitted per query and offset in Simple LSH. In contrast, the shuffle size for Layered LSH remains almost constant with increasing L and offers a factor 100 improvement over Simple LSH for the Random data set and on average a factor 50 improvement for the Image data set. This observation verifies Theorem 8, as well as the surprising property that network load of Layered LSH is independent of L, as described in Remark 9.

Although minimizing the "wall-clock" running time for the MapReduce implementation was not one of our primary objectives, we observe that improvement in the shuffle size also leads to a factor 4 improvement on average in run time for the Random data set and a factor 2 improvement on average for the Image data set, even with a crude binary search for D, as shown in Figure 4.2. Hence, this verifies load balancing properties of Layered LSH (Theorem 11). Note that since Hadoop offers checkpointing guarantees where (Key, Value) pairs output by mappers and reducers may be writ-



(a) Random: runtime (b) Image: runtime, $r = 0.08$ (c) Image: runtime, $r = 0.16$

Figure 4.2: Variation in "wall-clock" runtime with increasing L. Layered LSH yields a on average, a factor 4 improvement for the Random data set and a factor 2 improvement for the Image data set.

ten to the disk, Layered LSH also decreases the amount of data written to the distributed file system.

Sensitivity to variation in D:
Figure 4.3 shows the variation of shuffle size and run time with parameter D when $L = 200$ for both data sets in Fig 4.3. We observe that while the shuffle size decreases uniformly with increasing D, the "wall-clock" run time can be minimized by an appropriate choice of D. This "optimum" value of D represents the exact point where benefits of decreasing network cost of the distributed system match the disadvantages of sending near-by points to the same reducer. It is not clear to us whether a formal model of variation of network costs and load balance with D can be studied independent of the distributed framework and the underlying network topology. We leave its existence and formulation to future work. However, it is worth pointing out that typical deployments of this system would be for batch jobs that run repeatedly on MapReduce, or real-time jobs that run over long periods of time on Active DHT frameworks, so it is reasonable to assume a crude learning of the appropriate D.

Network calls are often the bottleneck for large scale applications which require to processing huge batches queries, or real time systems where queries arrive at a high rate and experiments presented in this section show that Layered LSH may lead to large improvement in such settings.

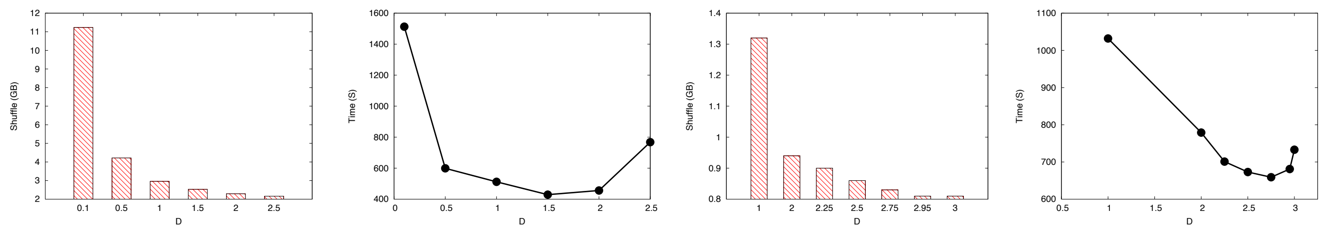
5. RELATED WORK

The methods proposed in the literature for similarity search via nearest neighbors (NN) can be, broadly speaking, classified to data or space partitioning methods and Locality Sensitive Hashing (LSH) techniques. Space partitioning methods, such as K-D trees [5], and data partitioning methods, such as R-trees [20] and SR-trees [24], solve the NN problem by pruning the set of candidates for each query using branch and bound techniques. However, they do not scale well with the data dimension, and can in fact be even slower than a brute force sequential scan of the data set when the dimension exceeds 10 [35]. For further details on these methods the reader is referred to the survey by Fodor [18].

LSH indexing [23] scales well with data dimension and is based on LSH families of hash functions for which near points have a higher likelihood of hashing to the same value. Then, near neighbors are retrieved by using multiple hash tables. Gionis et al. [19] showed that in the Euclidian space $O(n^{1/c})$ hash tables suffice to get a c-approximate near neighbor. This was later improved, by Data et al. [15], to $O(n^{\beta/c})$ (for some $\beta < 1$), and further, by Andoni and Indyk [4], to $O(n^{1/c^2})$ which almost matches the lower bound proved by Motwani et al. [30]. LSH families are also known for several non-Euclidian metrics, such as Jaccard distance [8] and cosine similarity [10].

The main problem with LSH indexing is that to guarantee a good search quality, it requires a large number of hash tables. This entails a large index space requirement, and in the distributed setting, also a large amount of network communication per query. To mitigate the space inefficiency, Panigrahy [32] proposed Entropy LSH which, by also looking up the hash buckets of $O(n^{2/c})$ random query "offsets", requires just $\tilde{O}(1)$ hash tables, and hence provides a large space improvement. But, Entropy LSH does not help with and in fact worsens the network inefficiency of conventional LSH: each query, instead of $O(n^{1/c})$ network calls, one per hash table, requires $O(n^{2/c})$ calls, one per offset. Our Layered LSH scheme exponentially improves this and, while guaranteeing a good load balance, requires only $O(\sqrt{\log n})$ network calls per query.

To reduce the number of offsets required by Entropy LSH, Lv et al. [29] proposed the Multi-Probe LSH (MPLSH) heuristic, in which a query-directed probing sequence is used instead of random offsets. They experimentally show this heuristic improves the number of required offset lookups. In a distributed setting, this translates to a smaller number of network calls per query, which is the main goal in Layered LSH as well. Clearly, Layered LSH can be implemented by using MPLSH instead of Entropy LSH as the first "layer" of hashing. Hence, the benefits of the two methods can be combined in practice. However, the experiments by Lv et al. [29] show a modest constant factor reduction in the number of offsets compared to Entropy LSH. Hence, since Layered LSH, besides a theoretical exponential improvement, experimentally shows a factor 100 reduction in network load compared to Entropy LSH, it is not clear if the marginal benefit from switching the first layer to MPLSH is significant enough to justify its much more complicated offset generation. Furthermore, MPLSH has no theoretical guarantees,



(a) Random: shuffle size (b) Random: run time (c) Image, $r = 0.08$: shuffle (d) Image, $r = 0.08$: run time

Figure 4.3: Variation in shuffle size and the "wall-clock" run time with parameter D for the Random (figures (a), (b)) and Image (figures (c), (c)) data sets.

while using Entropy LSH as the first layer of hashing allows for the strong theoretical guarantees on both network cost and load balance of Layered LSH proved in this paper. Hence overall, in this paper, we focus on Entropy LSH as the first layer of hashing in Layered LSH.

Haghani et al. [21] and Wadhwa et al. [34] study implementations of LSH via distributed hash tables on p2p networks aimed at improving network load by careful distribution of buckets on peers. However, they only provide heuristic methods restricted to p2p networks. Even though the basic idea of putting nearby buckets on the same machines is common between their heuristics and our Layered LSH, our scheme works on any framework in the general distributed (Key, Value) model (including not only MapReduce and Active DHT but also p2p networks) and provides the strong theoretical guarantees proved in this work for network efficiency and load balance in this general distributed model.

6. CONCLUSIONS

We presented and analyzed Layered LSH, an efficient distributed implementation of LSH similarity search indexing. We proved that, compared to the straightforward distributed implementation of LSH, Layered LSH exponentially improves the network load, while maintaining a good load balance. Our analysis also showed that, surprisingly, the network load of Layered LSH is independent of the search quality. Our experiments confirmed that Layered LSH results in significant network load reductions as well as runtime speedups.

7. REFERENCES

[1] Hadoop. http://hadoop.apache.org.

[2] http://tech.backtype.com/preview-of-storm-the-hadoop-of-realtime-proce.

[3] Tiny images dataset. http://horatio.cs.nyu.edu/mit/tiny/data/index.html.

[4] A. Andoni and P. Indyk. Near optimal hashing algorithms for approximate nearest neighbor in high dimensions. FOCS '06.

[5] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[6] P. Berkhin. *A Survey of Clustering Data Mining Techniques*. Springer, 2002.

[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbours. ICML '06.

[8] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. STOC '98.

[9] J. Buhler. Efficient large scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[10] M. Charikar. Similarity estimation techniques from rounding algorithms. STOC '02.

[11] M. Covell and S. Baluja. Lsh banding for large-scale retrieval with memory and recall constraints. ICASSP '09.

[12] T. Cover and P. Hart. Nearest neighbour pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[13] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filetering. WWW '07.

[14] S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, 2003.

[15] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality sensitive hashing scheme based on p-stable distributions. SoCG '04.

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI '04.

[17] T. Deselaers, D. Keysers, and H. Ney. Features for image retrieval: An experimental comparison. In *Information Retrieval*, volume 11, pages 77–107. Springer, 2008.

[18] I. K. Fodor. A survey of dimension reduction techniques. In *Technical Report UCRL-ID-148494, Lawrence Livermore National Laboratory*, 2002.

[19] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. VLDB '99.

[20] A. Guttman. R-trees: a dynamic index structure for spatial searching. SIGMOD '84, pages 47–57.

[21] P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. EDBT '09.

[22] S. Har-Peled. A replacement for voronoi diagrams of near linear size. FOCS '01.

[23] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. STOC '98.

[24] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. SIGMOD '97.

[25] R. Krauthgamer and J. Lee. Navigating nets:simple algorithms for proximity search. SODA '04.

[26] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. ICCV '09.

[27] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search of approximate nearest neighbor in high dimensional spaces. STOC '98.

[28] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. SIGMOD '11, pages 985–996.

[29] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. VLDB '07.

[30] R. Motwani, A. Naor, and R. Panigrahi. Lower bounds on locality sensitive hashing. SoCG '06.

[31] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. KDCloud '10.

[32] R. Panigrahi. Entropy based nearest neighbor search in high dimensions. SODA '06.

[33] D. Ravichandran, P. Pantel, and E. Hovy. Using locality sensitive hash functions for high speed noun clustering. ACL '05.

[34] S. Wadhwa and P. Gupta. Distributed locality sensitivity hashing. CCNC '10, pages 1 –4.

[35] R. Weber, H. Schek, and S. Blott. A quantititative analysis and performance study for similarity search methods in high dimensional spaces. VLDB '98.