# LoSHa: A General Framework for Scalable Locality Sensitive Hashing

Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, Ruihao Zhao
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{jfli,jcheng,fyang,yzhuang,yjzhao,xyan,rhzhao}@cse.cuhk.edu.hk

## ABSTRACT

Locality Sensitive Hashing (LSH) algorithms are widely adopted to index similar items in high dimensional space for approximate nearest neighbor search. As the volume of real-world datasets keeps growing, it has become necessary to develop distributed LSH solutions. Implementing a distributed LSH algorithm from scratch requires high development costs, thus most existing solutions are developed on general-purpose platforms such as Hadoop and Spark. However, we argue that these platforms are both hard to use for programming LSH algorithms and inefficient for LSH computation. We propose LoSHa, a distributed computing framework that reduces the development cost by designing a tailor-made, general programming interface and achieves high efficiency by exploring LSH-specific system implementation and optimizations. We show that many LSH algorithms can be easily expressed in LoSHa's API. We evaluate LoSHa and also compare with general-purpose platforms on the same LSH algorithms. Our results show that LoSHa's performance can be an order of magnitude faster, while the implementations on LoSHa are even more intuitive and require few lines of code.

## 1 INTRODUCTION

*Nearest neighbor* (*NN*) search, which finds items similar to a given query item, serves as the foundation of numerous applications such as entity resolution, de-duplication, sequence matching, recommendation, similar item retrieval, and event detection. However, many indexing methods for NN search are inefficient when data dimensionality increases, and eventually perform even worse than brute-force linear scans [20]. Unfortunately, many types of data in real world, including texts, images, audios, DNA sequences, etc., are expressed and processed as high dimensional vectors.

To find similar items in high dimensional space, recent solutions focus on finding *approximate nearest neighbors* (*ANN*), for which *Locality Sensitive Hashing* (*LSH*[1]) [13] is widely recognized as the most effective method [20, 31]. The idea of LSH is that, by using carefully designed hash functions, similar items are hashed to the same bucket with higher probability and one only needs to check a small number of buckets for items similar to a query. LSH can be implemented on relational databases and has sub-linear querying time complexity regardless of the distribution of data/queries [31]. Due to its outstanding query performance and favorable characteristics, LSH algorithms have been extensively studied [7, 8, 12, 14, 19, 31, 32, 36]. There are also numerous applications of LSH in computer vision, machine learning, data mining and database. Due to its significance, LSH is cited as one of the two pieces of "Breakthrough Research" (the other one being MapReduce) by the 50th Anniversary issue of Communications of the ACM.

As we will discuss in Section 2.2.1, LSH algorithms are either *single-probing* or *multi-probing*. Query processing with single-probing LSH is simple but requires hundreds of hash tables in order to achieve good accuracy [30], which results in high memory and CPU usage. LSH algorithms developed in the early days, such as MinHash [2], SimHash [3] and E2LSH [5], belong to this category. In recent years, multi-probing LSH algorithms were proposed to reduce the number of hash tables [7, 12, 20, 24, 29, 31, 36], but the tradeoff is that query processing also becomes more complicated.

LSH is usually used to handle large-scale data (e.g., billions of images [29] or tweets [30]), for which distributed LSH has become necessary (to address the resource limitation of a single machine). However, implementing *distributed multi-probing LSH* is challenging due to the following two main reasons.

First, query processing with multi-probing LSH is complicated to be distributed. In fact, existing multi-probing LSH algorithms mainly adopt external-memory implementations [7, 12, 20, 29, 31, 36], which however have limited query throughput and long delay due to the computing capacity of a single machine.

Second, there is a lack of a general programming framework to implement LSH algorithms on existing general-purpose distributed frameworks such as Hadoop and Spark. While there are works such as Pregel [25] and PowerGraph [10] provide a general programming framework for graph processing [23], and parameter servers [15] for machine learning, there is no such framework for LSH implementations. For this reason, most of the distributed solutions [1, 18, 21, 22, 27, 28, 30] are single-probing LSH only, whose query processing is simple and easy to implement.

To address the above problems, we set out to design a general platform, called **LoSHa**, for LSH workloads with tailor-made abstractions and user-friendly APIs. LoSHa offers a simple `query-answer` programming paradigm (resembling map-reduce) so that users can express different types of LSH algorithms (especially multi-probing LSH) easily and concisely, while implementation details such as

[1]In this paper, LSH refers a broad class of algorithms [14, 32] and will be discussed in Section 2.

*SIGIR '17, August 07-11, 2017, Shinjuku, Tokyo, Japan*
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5022-8/17/08…$15.00
http://dx.doi.org/10.1145/3077136.3080800

defining complicated dataflow, choosing/implementing different operators (e.g., joins, de-duplication), workload distribution, and fault tolerance, are all shielded from users.

We implemented LoSHa as a subsystem upon a general-purpose platform (i.e., Husky [16, 33, 34]) to demonstrate its feasibility. We explored LSH specific data layout and system optimizations, and evaluated LoSHa on large real datasets, including 1 billion tweets and 1 billion image descriptors. Our results show that LoSHa is over an order of magnitude faster than existing LSH implementations [1, 28] on Spark and Hadoop, and can handle 10 times larger datasets with the same computing resources.

## 2 BACKGROUND

In this section, we give the background of LSH and highlight the challenges of making LSH applications distributed.

### 2.1 Notion and Notations of LSH

Indyk and Motwani [13] introduced the notion of LSH as follows.

*Definition 2.1.* Let $B(q, r)$ denote a ball centered at a point $q$ with radius $r$. A hash function family $\mathcal{H} = \{h : R^d \to U\}$ is $(r_1, r_2, p_1, p_2)$-sensitive, if given any $q, o \in R^d$,

(1) if $o \in B(q, r_1)$, then $Pr_{\mathcal{H}}[h(o) = h(q)] \geq p_1$;
(2) if $o \notin B(q, r_2)$, then $Pr_{\mathcal{H}}[h(o) = h(q)] \leq p_2$.

A useful LSH family should satisfy inequality $r_1 < r_2$ and $p_1 > p_2$. For any $d$-dimensional point $o \in R^d$, a single hash function $h \in \mathcal{H}$ can hash $o$ to a value $v \in U$. Two similar points with distance less than or equal to $r_1$ have a high probability of at least $p_1$ to be hashed to the same value, while two dissimilar points with distance larger than $r_2$ have a low probability of at most $p_2$ to be hashed to the same value. Some common LSH families include *min-wise permutation* for Jaccard distance [2], *random projection* for angular distance [3], and 2-*stable projection* for Euclidean distance [5], also known as *MinHash*, *SimHash*, and *E2LSH*, respectively.

Intuitively, a hash value $v \in U$ represents a bucket that contains similar items (i.e., points). We denote the probability of two items being hashed to the same value as the *to-same-bucket* probability. To improve the accuracy of LSH, multiple hash tables are generated, and multiple hash values are concatenated and used as a whole for each hash table. Specifically, $L \geq 1$ hash tables, i.e., $G_1, G_2, \ldots, G_L$, are generated. For each hash table, $k \geq 1$ randomly chosen hash functions, $(h_1, h_2, \ldots, h_k)$, are used, and the $k$ hash values of an item $o$ are concatenated to form a *signature*, $G(o) = (h_1(o), h_2(o), \ldots, h_k(o))$. In other words, $L$ signatures, i.e., $G_1(o), G_2(o), \ldots, G_L(o)$, are obtained for the item $o$. The purpose of using $k$ hash functions and $L$ hash tables is to enlarge the gap of to-same-bucket probability between similar items and dissimilar items [27].

LSH is typically conducted in two phases. The first phase is *pre-processing*, which inserts data items into the hash tables. LSH projects each $d$-dimensional data item into a $k$-dimensional vector, i.e., a $k$-dimensional signature $G_i(o)$, and inserts the item into the corresponding bucket in hash table $G_i$, for $1 \leq i \leq L$, where $G_i(o)$ is the bucket ID. The second phase is *querying*. Given a *query item* $q$, we first obtain $L$ bucket IDs for $q$ in the same way as for a data item. Then, data items in the same bucket as $q$ (called *candidate answers*) in each hash table are evaluated to obtain the final answer (e.g., items within a given distance from $q$). LSH achieves sub-linear querying complexity [13, 31] (as only candidate answers are evaluated) and hence is attractive in practice.

### 2.2 The State-of-the-art LSH

We first categorize existing LSH algorithms and then analyze the difficulties of implementing them on existing distributed frameworks.

*2.2.1 LSH Algorithms.* Querying by LSH was originally designed to probe a single bucket (i.e., the bucket $q$ is hashed to) in each of the $L$ hash tables. We name this querying method as *single-probing LSH* and typical examples include MinHash [2], SimHash [3] and E2LSH [5]. Single-probing LSH is simple [1, 18, 21, 22, 27, 28], but it suffers from two weaknesses. First, single-probing LSH is sensitive to $r_1$ and $r_2$, but the distance from a query to its nearest neighbor could be any number in practice. To guarantee a high recall (i.e., most of the query answer are found in the set of candidate answers), a large number of hash tables have to be generated [24, 31], which consumes a large amount of memory and incurs high deduplication overhead to remove duplicate candidate answers from the $L$ buckets. For example, 780 hash tables are created in PLSH [30]. Second, single-probing LSH has poor recall if $k$ is large (e.g., 64 or 128) [14, 19]. Thus, it is not applicable to most algorithms [14, 32] that learn hash functions from data and map each item to a hash code of long bits for better precision [19].

To address the weaknesses of single-probing LSH, many *multi-probing LSH* algorithms have been proposed [7, 12, 20, 24, 29, 31, 36]. The main idea is that, if similar items are not in the same bucket as $q$, they are likely in other buckets in the hash table that are "nearby" $q$'s bucket because *the hashing is locality sensitive.* If we can compute new signatures corresponding to these "nearby" buckets, we can probe multiple buckets in the same hash table to improve the recall. As a result, multi-probing significantly reduces the number of hash tables [7, 12, 20, 24, 36] and effectively improves the recall when $k$ is large [19, 32].

*2.2.2 Distributed LSH.* Although there are many LSH algorithms, no one can dominate others in all aspects (e.g., precision, recall and efficiency) [14, 26, 32]. And one common problem they have is the handling of large-scale data (e.g., billion-scale tweets or images), for which the typical solution is by distributed computing. To reduce the development cost, general-purpose distributed frameworks are normally used and most of distributed LSH solutions were developed on Hadoop and Spark. However, these distributed solutions are mostly single-probing LSH [1, 18, 21, 22, 27, 28] and suffer from the problems presented in Section 2.2.1. In addition, it is also challenging to implement distributed multi-probing LSH on Hadoop and Spark, due to the following two main factors.

First, to program multi-probing LSH, users need to define a complicated dataflow consisting of many steps. Each step involve multiple operations, and users have to carefully select suitable operators (e.g., which join operator to use) and consider many other implementation details (e.g., how to avoid unnecessary recomputation by data caching on Spark). It is easy to make a wrong choice in some steps, leading to an inefficient or even incorrect

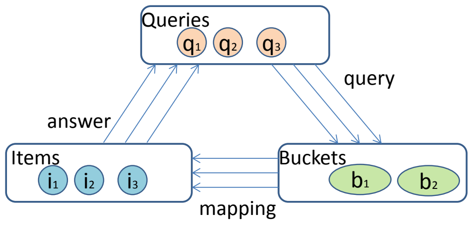


**Figure 1: Query processing in LoSHa**

implementation, much debugging and performance tuning efforts. To address these problems, we design higher-level abstractions and user-friendly APIs in Section 3, and demonstrate with examples in Section 4 how to implement multi-probing LSH algorithms.

Second, existing general-purpose platforms are not optimized for LSH. For example, in the LSH implementation on these platforms, queries and items are joined on signatures, which means that $L$ copies of items and queries need to be shuffled and sent over the network, leading to high network traffic and enormous memory consumption. Furthermore, the execution strategies of their operators (e.g., join) are not best fit for LSH workloads. To give efficient implementations, users need to do the optimizations by themselves. In Section 5, we explore LSH-specific optimizations so that users are off-loaded from these difficult tasks.

## 3 LOSHA PROGRAMMING FRAMEWORK

A practical distributed computing framework must provide an easy-to-use programming interface in order to lower the development cost. One well-known example is MapReduce [6], which offers a simple programming interface for easy implementation of many distributed algorithms for data processing. Similarly, for scalable LSH computation, the most important first step is to design a unified programming model and a simple API. However, this is challenging because there are hundreds of LSH algorithms and many of them are complicated to implement, especially parallelizing them. We present the details of the LoSHa programming model in this section.

### 3.1 Overall Framework

LoSHa uses three data abstractions, `Query`, `Bucket`, and `Item`, to model input queries, buckets, and items, respectively. LoSHa first pre-processes data items to create a set of `Item` objects. It also creates a set of `Bucket` objects, where each `Bucket` object has a bucket ID, `bId`, and all `Item` objects having a signature equal to `bId` belong to this `Bucket` object. Queries are also processed similarly as items to create a set of `Query` objects.

Query processing in LoSHa requires users to specify the querying logic in two user-defined functions, `query` and `answer`. Figure 1 shows the flow of query processing, described as follows. LoSHa processes LSH queries iteratively. In each iteration, each `Query` object calls the `query` function to find a set of *potential buckets* that contain candidate items, and sends a message `QueryMsg` to each potential bucket, where `QueryMsg` contains the content of the query (and any other information required to evaluate the query). When a `Bucket` object receives a `QueryMsg` message, it forwards the message to its `Item` objects. When an `Item` object receives a `QueryMsg` message, it calls the `answer` function to evaluate the query (e.g.,

**Listing 1** Query API in LoSHa

```
template<typename Id,
    typename Content,
    typename QueryMsg = IdContent<Id, Content>,
    typename AnswerMsg = std::pair<Id, float>>
class Query
  :public IdContent<Id, Content>{
public:
  virtual void query(Factory<Id, Content>& fty);

  const IdContent<Id, Content>& getQuery();

  const std::vector<AnswerMsg>& getAnswerMsg();

  void sendToBucket(Sig& bId, QueryMsg& msg);

  void setFinished();
};
```

calculate the distance between the query and itself), and sends the results (as an `AnswerMsg` message) back to the `Query` object. Then in the next iteration, each `Query` object decides whether the query evaluation is completed or should continue based on the results received from the candidate `Item` objects.

Iterative processing allows multi-probing LSH to be easily implemented; while for conventional single-probing LSH, we simply run just one iteration. LoSHa hides the complicated dataflow in the iterative process from users. The underlying system also optimizes key operations such as join and de-duplication, and automatically handles workload distribution and fault tolerance.

### 3.2 Application Programming Interface

In LoSHa, `Query`, `Bucket` and `Item` objects and their related operations are implemented by the respective `Query`, `Bucket` and `Item` classes in LoSHa's API. To program with LoSHa, users only need to subclass the `Query` and `Item` classes, and overload the `query` and `answer` functions.

The APIs for `Query` and `Item` are shown in Listings 1 and 2. Users specify four template arguments, `Id`, `Content`, `QueryMsg` and `AnswerMsg`. LoSHa expresses a query or an item by an `(Id, Content)` pair, denoted by `IdContent`, where `Id` is the type of the ID of the query/item and is used to locate the query/item, and `Content` is the type of the content of the query/item. LoSHa stores `Content` in a vector. The $i$-th element can be either the value of the $i$-th feature of the query/item (i.e., dense vector), or a $(pos, v)$ pair representing the value $v$ of the $pos$-th feature (i.e., sparse vector). The messages sent out by a query and an item are of type `QueryMsg` and `AnswerMsg`, respectively. For example, `QueryMsg` can be the query itself, i.e., `IdContent`, or just the query ID. An `AnswerMsg` message sent by an item is usually an `(Id, dist)` pair, where `Id` is the item ID and `dist` is the distance between the query and the item. A `Query` or `Item` object may also contain user-defined fields to carry any information useful for query processing.

**Query API.** Users implement the `query` function to specify the operations to be performed by a `Query` object (see an example in Listing 3). The `Query` class provides a number of built-in functions, and the `query` function may use `getQuery` to obtain a query's `IdContent`, use `getAnswerMsg` to collect messages sent by candidate items, and use `setFinished` to finish the evaluation of a query when the query predicate is satisfied. Signatures of a query

**Listing 2** `Item` API in LoSHa

```
template<typename Id,
    typename Content,
    typename QueryMsg = IdContent<Id, Content>,
    typename AnswerMsg = std::pair<Id, float>>
class Item
  :public IdContent<Id, Content>{
public:
  virtual void answer(Factory<Id, Content>& fty);

  const IdContent<Id, Content>& getItem();

  const std::vector<QueryMsg>& getQueryMsg();

  void sendToQuery(Id& qId, AnswerMsg& msg);
};
```

**Listing 3** Single-probing LSH in LoSHa

```
class LSHQuery : public Query<int, float>{
public:
  void query(Factory<int, float>& fty){
    //obtain buckets (i.e., signatures) to probe
    for(auto& bId : fty.calSigs(*this))
      sendToBucket(bId, this->getQuery());
  }
};
class LSHItem : public Item<int, float>{
public:
  void answer(Factory<int, float>& fty){
    //get messages to evaluate
    for(auto& q: getQueryMsg()){
      float dist = fty.calDist(q, *this);
      //output qualified items
    }
  }
};
```

can be calculated by functions provided in the `Factory` API (to be presented shortly). Each signature corresponds to one bucket ID, which is of type `Sig` (a $k$ dimensional vector in LoSHa by default), and `sendToBucket` uses the ID to send messages to the bucket.

**Item API.** Users may use `getQueryMsg` in the `Item` class to collect messages forwarded to the item by `Bucket` objects, and then specifies in the `answer` function how to process the messages to evaluate the query (e.g., calculate the distance between the item and the query). The results are then sent via an `AnswerMsg` message to the corresponding `Query` object by `sendToQuery`.

**Factory API.** Many types of metric spaces are defined for different LSH applications. A different metric space requires a different distance calculation and signature computation. The `Factory` API allows three user-defined functions to be overloaded. Users can overload the `calDist` function to calculate the distance between two points in the given metric space, and the `calSigs` function to compute the signatures for a given point. Some LSHs (e.g., E2LSH [5]) first project an item to a real value and then round it to an integer to obtain the final hash value for the item. Rounding loses proximity information, which is useful for designing some multi-probing LSH strategy. Thus, we provide another function, `calProjs`, for users to obtain the un-rounded projected real values of a query or an item.

LoSHa supports a `Factory` library which includes many common metrics such as Hamming distance, Jaccard distance, Euclidean distance, Angular distance, etc. For a wide range of LSH algorithms, users may simply call functions for their applications using any of these metrics, and focus on the design of the querying logic. Users may also implement their own `Factory` functions for a new metric.

**An example.** We illustrate how to program with LoSHa's API by implementing a single-probing LSH algorithm, as shown in Listing 3. The `query` function calls `calSigs` of the `Factory` class to compute the signatures of a query, i.e., the IDs of the potential buckets to be probed, and then calls `sendToBucket` to send the `QueryMsg` message to each potential bucket. The `QueryMsg` message is just the query itself in this application.

The `answer` function simply checks each `QueryMsg` message forwarded to an item by its bucket, and calls the `calDist` function of the `Factory` class to calculate the distance between the item and the query in `QueryMsg`. The qualified answers are then written to the output.

The single-probing LSH implementation is simple. For multi-probing LSH, we only need to specify the multi-probing strategy in the `query` function, which we discuss in the following subsection, while in the `answer` function we simply use `sendToQuery` to send the results back to the query (instead of writing to the output).

## 3.3 Multi-probing LSH Strategies

Many multi-probing LSH algorithms have been proposed for various application scenarios [7, 20, 29, 31]. In order to provide a general programming framework for LSH implementations, we identify common patterns in these algorithms. Specifically, in the first iteration, we probe buckets as in single-probing. If the query predicate is not satisfied (e.g., less than $K$ similar items are found), new signatures (i.e., new bucket IDs) are generated according to a multi-probing strategy, and we continue probing these new potential buckets in the next iteration. To demonstrate our framework can support multiple probings well, we adopt two strategies in the experiments and present them below.

**Candidate expansion using similar items.** The main idea of this strategy is that similar items are more likely to be clustered together. Assume $o$ is a similar item of $q$ returned in the current iteration from the $l$-th hash table, where $1 \leq l \leq L$. For each $o$, the `query` function computes a set of potential buckets (for the next iteration) with IDs equal to $G_1(o), \ldots, G_{l-1}(o), G_{l+1}(o), \ldots, G_L(o)$. Since $o$ is returned from the bucket with ID $G_l(o)$ in the $l$-th hash table, we do not probe this bucket again to avoid duplicate probing. This strategy is good for radius-based search (i.e., return neighbors within a specific distance) as shown in Section 6.2.

**Adjusted signatures.** Many LSH algorithm [19, 20, 24] will first hash a point to a real value, then truncate it to an integer. The truncation may lose some proximity information between two points. For example, an E2LSH function [5] may hash two point $q$ and $o$ to values 1.001 and 0.999, which are then truncated to 1 and 0, respectively. Although the difference between $q$ and $o$ is only 0.002, it is enlarged by the truncation to 1. Such loss can be dismissed by adjusting the hash value of $q$ from 1 to 0, and the idea has been used to design multi-probing LSH strategies in [24]. We implement the idea in LoSHa as follows. Let $r_i(q)$ be the real value that is truncated to give $h_i(q)$, where $1 \leq i \leq k$. We first calculate $d_i(q) = r_i(q) - h_i(q)$, sort $d_i(q)$ for $1 \leq i \leq k$ in ascending order of their values, and keep the sorted list in $q$'s data field. Then, in the $i$-th iteration after the first iteration, if the query predicate is

**Listing 4** Bucket API in LoSHa

```
template<typename Id,
         typename Content,
         typename QueryMsg,
         typename AnswerMsg>
class Bucket{
public:
  virtual void mapping(Factory<Id, Content>& fty);

  const std::vector<QueryMsg>& getQueryMsg();

  void sendToQuery(Id& qId, AnswerMsg& msg);
};
```

not satisfied, let $d_j(q)$ be the $i$-th value in the sorted list, the `query` function computes a new potential bucket ID by adjusting the $j$-th hash value in $q$'s signature. This strategy significantly improves the quality of returned answers for $k$-nearest-neighbor search as reported in Section 6.4.

## 3.4 Support for Non-Querying Applications

Some applications of LSH do not require a querying process. For example, for overlapping clustering applied in Google News [4], we can implement it in MapReduce by first computing the signatures of the items in mappers and then distributing the items to reducers based on their $L$ signatures. Each reducer then simply outputs the items distributed to it, labeling them as in the same cluster. Although LoSHa is a framework designed for query processing, we show that LoSHa can also handle such a non-querying workload as easily as MapReduce using the `Bucket` API given in Listing 4.

We consider each item as a query so that `Query` acts as `Mapper` and the `query` function acts as the `map` function, while the `Item` class is simply not used here. We also subclass the `Bucket` class, which acts as `Reducer`, and the `mapping` function of `Bucket` acts as the `reduce` function. In addition, our framework also supports efficient implementation of iterative algorithms which are expensive when implemented on the MapReduce framework.

## 4 APPLICATIONS

In this section, we illustrate how to implement two advanced multi-probing LSH algorithms in LoSHa's framework.

## 4.1 Multi-Probing PLSH in LoSHa

PLSH [30] is the fastest distributed LSH implementation for angular distance, but it is single-probing LSH and requires a large number of hash tables. We implement the LSH algorithm of PLSH with a multi-probing strategy to reduce the number of hash tables. We denote our implementation as *multi-probing PLSH* (*MPLSH*).

Implementing the `query` function for multi-probing LSH consists of three main steps: *(1) collecting candidates from the previous iteration, (2) deciding whether to finish querying, (3) computing a new set of potential bucket IDs for the next iteration.* In Listing 5, the `query` function performs the above three steps as follows. First, it collects the candidate items sent from the previous iteration, if any, into a container *cands*. Then, the candidate items in *cands* are used to check whether the query predicate is satisfied (e.g., top $K$ answers have been found), and to decide whether to finish or continue the query evaluation (note that users should output the query results before calling `setFinished`). If the query predicate is not satisfied yet, a new set of signatures is then calculated (e.g., using a probing strategy described in Section 3.3) and `sendToBucket` is called to initiate probing the potential buckets for the next iteration.

**Listing 5** Multi-probing PLSH in LoSHa

```
class MPLSHQuery
  : public Query<int, Content, QueryMsg, int>{
public:
  virtual void query(Factory<int, Content>& fty){
    //collect candidates into cands
    for(auto& item : getAnswerMsg() )
      cands.insert(item);
    auto finished = ... //decide whether to stop
    if(finished) setFinished();
    //calculate new buckets(signatures) to probe
    auto newSigs = ... //calculate new signatures
    for(auto& bId : newSigs)
      sendToBucket(bId, this->getQuery());
  }
};
class MPLSHItem
  : public Item<int, Content, QueryMsg, int>{
public:
  void answer(Factory<int, Content> &fty){
    for(auto& q: getQueryMsg()){
      float dist = fty.calDist(q, *this);
      if(dist <= 0.9) sendToQuery(q.id, this->id);
    }
  }
};
```

The `answer` function is much simpler. We only need to calculate the distance between a candidate item and a query using a standard angular distance calculation function in the `Factory` class, and calls `sendToQuery` to send the qualified candidates (e.g., items having an angle smaller than 0.9 from the query) to the query.

## 4.2 Parallelizing C2LSH with LoSHa

The problem of $c$-ANN [7] returns a data item $o$ for a query $q$, such that $d(o, q) \leq c \cdot d(o^*, q)$, where $o^*$ is the exact NN of $q$, $d(o, q)$ and $d(o^*, q)$ denote the distance between $o$ and $q$, and between $o^*$ and $q$. Many multi-probing LSH solutions have been proposed for this problem, but most are external-memory algorithms [7, 20, 29, 31, 36]. With our framework, we can easily implement a parallel version of these multi-probing LSH algorithms, which utilizes the aggregate memory of a cluster instead of resorting to the external memory of a single machine. We illustrate by converting C2LSH [7], one of the most efficient single-core multi-probing external-memory LSH algorithms, into a distributed LSH algorithm with LoSHa.

C2LSH works in Euclidean distance with E2LSH [5] as the hash function family. Only one hash function is used to compute each signature, i.e., $k = 1$, and hence we simply use $h(.)$ for $G(.)$. The `query` function probes bucket $h(q)$ in the first iteration, and then in the $(i + 1)$-th iteration (for $i > 0$), buckets with ID $bId$ satisfying $\lfloor bId/(c^i) \rfloor = \lfloor h(q)/(c^i) \rfloor$ are probed. In the $i$-th iteration, the `query` function calls `setFinished` if it has collected more than $K$ candidates whose distance to $q$ is less than or equal to $c^i$, or more than $(K + \beta n)$ candidates are returned, where $\beta$ is a user-defined percentage of false positives that are allowed and $n$ is the total number of items. In C2LSH, an item $o$ is considered as a candidate only if it collides with $q$ in more than $\theta$ probed buckets. Thus, in the `answer` function, the collision count of an item with each query is maintained (as the data field of the item). If the count is larger than $\theta$ for a query $q$, the item will be sent to $q$.

Readers may refer to [7] for theoretical guarantees. We remark that the implementation of C2LSH on LoSHa takes about 200 lines of code, while the code of [7] has over 3,000 lines in order to handle large datasets.

## 5 SYSTEM IMPLEMENTATION

We now present the implementation details of LoSHa.

### 5.1 System Architecture

LoSHa is designed to run on a shared-nothing cluster with a typical master-worker architecture. The master mainly performs progress checking and coordination on the workers. Each machine in the cluster may run multiple workers, which perform the LSH computation on data.

A LoSHa program proceeds in iterations, where each iteration consists of three steps: `query`, `mapping`, and `answer`. A worker is processed by one thread. While all workers run in parallel, each worker executes in sequence the `query`, `mapping`, and `answer` functions for the sets of queries, buckets, and items that are distributed to the worker. Workers communicate with each other by message passing, and each worker maintains $W$ message buffers, where $W$ is the number of workers in the cluster. For messages to be sent to workers in the same machine, they can be referred without actual payload copy and network communication.

### 5.2 Data Layout

Data items and queries are stored as `Item` and `Query` objects in LoSHa. LoSHa generates $L$ signatures for each item, using the `calSigs` function of the `Factory` class. Then, a `Bucket` object is created for each distinct signature (which is the ID of the bucket), and for all `Item` objects that have this signature, their item IDs are stored with this `Bucket` object. The `Item`, `Bucket` and `Query` objects are partitioned and distributed among workers by random hashing on their IDs.

LSH computation often requires to deliver messages to the receiving objects by their IDs. This can be implemented by hash join, but it incurs a higher overhead due to random access. We pre-sort objects in each worker by their IDs, and then sort messages by their target object IDs (if not sorted already) and hence the join requires only a linear scan. This allows better data locality and prefetching to improve performance. However, new queries keep coming in and there may also be new items inserted. To avoid incurring high cost in maintaining the pre-sorted order of the objects, we assign the ID of each new `Item`/`Query` object in increasing order as follows: worker $i$ assigns the $j$-th new object with ID $(i + j * W)$. In this way, we can simply append the new object to the end of the pre-sorted array without re-ordering.

LoSHa's data layout also allows it to implement an efficient two-level join that maps queries first to potential buckets and then to candidate items. In contrast, existing LSH implementations in MapReduce framework require expensive joins on queries and items, which have higher CPU and memory usage, and also higher data transmission cost among the workers, as verified by our experiments.

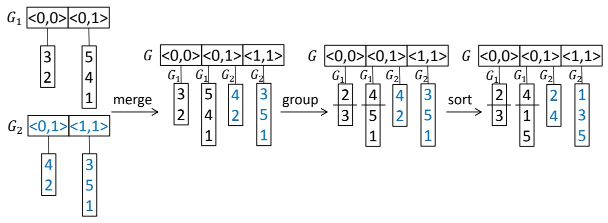


**Figure 2: Bucket compression**

### 5.3 Bucket Compression

Each item generates $L$ signatures, and each signature corresponds to a bucket in a hash table. Suppose that each hash table has $B$ distinct signatures on average. This requires $O(LBk)$ space for keeping the bucket objects and thus the memory consumption is high, because both $L$ and $B$ are large. For example, PLSH [30] uses $L = 780$ hash tables and $B = O(2^k)$ where $k = 16$. Note that PLSH is designed for SimHash, where each of the $k$ hash values in a signature is boolean, and $B$ can be much larger for other LSH families [5, 7, 12, 24, 29, 31] where each hash value is an integer.

We found that we can actually reduce the $O(LBk)$ space to $O(Bk)$ space by compressing the buckets with the same signature across the $L$ hash tables. Such a compression is effective based on our observation that the size of the union of the sets of signatures of the $L$ hash tables is $cB$, where $c$ is usually a small constant. Thus, we can just use one hash table with $cB$ distinct signatures, and keep all the item IDs with the same signature across the $L$ hash tables in one single location. As illustrated in Figure 2, we compress the buckets with the same signature (i.e., $< 0, 1 >$) in the two hash tables $G_1$ and $G_2$ into a single bucket in $G$. Then, we are able to apply further optimizations as follows.

For each list of item IDs kept in a bucket, we first group them by the workers where the real items are stored, and then sort each group. For example, if we have two workers and an item with ID $x$ is stored in worker $x\%2$, then the item IDs in Figure 2 are divided into two groups (i.e., even and odd) and then sorted. This new bucket structure has the following advantages. First, messages sent to multiple `Bucket` objects with the same signature are now sent to a single `Bucket` object. Second, to forward messages to items, we do not need to locate the items by hashing their IDs one by one, as the items are already grouped by their worker IDs and so we can simply forward the messages to their worker. Third, when their worker receives the messages (from multiple workers), we only need to merge them by the target item IDs which are already sorted (thus avoiding sorting during query processing).

### 5.4 Message Reduction

One major source of messages is the `QueryMsg` messages sent from each `Query` object to its potential buckets and then forwarded to all the items in each potential bucket. Since the `answer` function may calculate the distance between a query and a candidate item, the `QueryMsg` message should contain the query itself (i.e., a high dimensional vector) and thus sending `QueryMsg` messages many times through the network is expensive. To address this problem, we broadcast the query (together with other information needed for its evaluation) to each machine in the cluster, and only send query IDs in `QueryMsg`. Thus, when a worker executes the `answer` function on an item, it simply retrieves the corresponding query from the local machine. This significantly reduces the communication cost especially for multi-probing LSH.

Sending query IDs in `QueryMsg` and item IDs in `AnswerMsg` can be still costly since a large number of messages can be created in each step in an iteration. Combiner is a standard technique used to reduce the number of messages [6, 25]. To allow more message reduction, we implement a machine-level combiner. For the same `QueryMsg` message that is sent to multiple buckets in multiple workers, if these workers are in the same physical machine, LoSHa only sends a single `QueryMsg` message (together with the set of bucket IDs) to the machine. After reaching the remote machine, a thread is used to deliver the `QueryMsg` message to the corresponding buckets in each worker in this machine. In addition, all messages sent to the same machine are grouped together and sent in batches to make better use of network bandwidth.

In the `mapping` step in each iteration, however, we implement a two-level combiner because up to $L$ copies of the same `QueryMsg` message can be forwarded to the same item, i.e., the item appears in all the $L$ buckets to which the `QueryMsg` message is sent. We first combine the same `QueryMsg` messages sent to the same item at the worker level, and then we further combine the messages for the same item at the machine level. Finally, we combine the same `QueryMsg` messages that are sent to multiple items in multiple workers in the same target machine.

## 5.5 De-duplication

De-duplication is one of key issues in LSH implementations, as the same query can be mapped to the same item up to $L$ times. In addition, after the query has been evaluated against an item in an iteration, multi-probing may generate new potential bucket IDs in any subsequent iterations such that the same item may appear again in some of these new potential buckets, meaning that the query would be evaluated against the item again if de-duplication is not performed.

De-duplication is mainly done by using a hash set, tree index, or bit vector [30]. In LoSHa, most duplicate copies of the same `QueryMsg` message sent to the same item have already been eliminated by the two-level combiner described in Section 5.4, and the number of multiple copies of the same `QueryMsg` message sent to the item from different machines is usually very small. Thus, it is efficient to simply use a hash set to do the de-duplication.

LoSHa also differs from existing work in that we take an flexible approach to decide where de-duplication should be handled. The evaluation of a query against an item (e.g., distance calculation) can be expensive for some applications (e.g., each data item is a dense high dimensional vector), while it is cheap for other applications. If the evaluation is expensive, then de-duplication is performed by an `Item` object: the item inserts the received `QueryMsg` messages into a hash set, which removes all duplicate copies. However, keeping a hash set for each item may consume much memory and incur extra cost of deleting old queries. Thus, if the evaluation is not expensive, de-duplication is better performed by a `Query` object: we allow duplicate copies of a query to be evaluated against an item, but insert duplicate `AnswerMsg` messages received by the `Query` object into a hash set, so that duplicate qualified items will be removed and not included in the final answer.

**Table 1: Datasets**

| Dataset | Item# | Dim# | Entry# |
|---|---|---|---|
| `SIFT1B` | 1,000,000,000 | 128 | 128 |
| `Tweets1B` | 1,050,000,000 | 500,000 | avg 7.5 |
| `Glove2.2M` | 2,196,017 | 300 | 300 |

## 5.6 Online Processing and Fault Handling

LoSHa handles online queries and fault tolerance as follows.

**Online queries.** LoSHa is able to accept queries in real time. New queries will be read into a buffer in LoSHa. A greedy strategy can be used here such that each worker processes approximately the same number of queries. We adopt the mini-batch model as in PLSH [30], where a worker buffers new queries and starts their evaluation in the next iteration. Thus, new queries do not have to wait until all existing queries finish their evaluation.

**Fault tolerance.** Fault tolerance in LoSHa is by writing a copy of the item and bucket layout in each machine to HDFS. Usually we do infrequent batch-updates and hence we can re-write the entire copy of the item and bucket layout to HDFS each time. If the updates are frequent, we may apply checkpointing periodically. Query evaluation results are not backed up and we simply re-run the queries since the evaluation cost for a query is low.

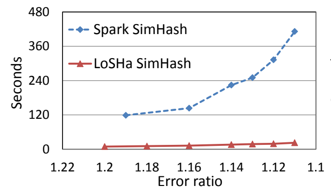
## 6 EXPERIMENTAL EVALUATION

We evaluated LoSHa, by comparing with specific LSH implementations on existing general-purpose platforms and a highly optimized system [30] for SimHash. We also assessed the effectiveness of various optimizations in LoSHa and tested its scalability. LoSHa was implemented in C++ and will be made open source on `http://www.husky-project.com/`.

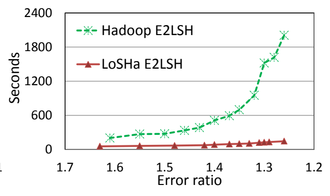
**Setup.** We ran the experiments on a cluster of 20 machines connected by a Broadcom's BCM5720 Gigabit Ethernet Controller. Each machine has two 6-core 2.0GHz Intel Xeon E5-2620 processors, 48GB DDR3-1,333 RAM, and a 600GB SATA disk (6Gb/s, 10k rpm, 64MB cache). CentOS 6.5 with 2.6.32 linux kernel and Java 1.8.0 are installed on each machine. We used Hadoop 2.6.0 and Spark 1.6.1. For Hadoop, we set the number of containers to 11 per machine (i.e., one for each core) and each is allocated 4GB RAM, and leave 1 core and 4GB RAM for other operations. For Spark, we set 4 workers per machine, and each worker is assigned 3 cores and 11GB RAM. We ran Spark on its standalone mode.For LoSHa, we used 12 cores per machine.

**Datasets.** We used three real datasets. For dense vectors, we used the largest publicly available dataset for ANN search, denoted by `SIFT1B`, which contains 1 billion SIFT image descriptors from http://corpus-texmex.irisa.fr/. We also used another dataset, denoted by `Glove2.2M`, which contains about 2.2 million vector representations for words obtained from http://nlp.stanford.edu/projects/glove/ (Common Crawl: 840B tokens, 2.2M vocab, cased, 300d vectors). For sparse vectors, we crawled 1.05 billion tweets from Twitter, denoted by `Tweets1B`. Table 1 lists the number of items (Item#), the number of dimensions of each item vector (Dim#), and the number of non-empty entries in each vector (Entry#).

For each dataset, we randomly selected 1,000 items as queries and removed these items from the datasets. Each query finds the top 10 ANNs of the query (unless otherwise specified).

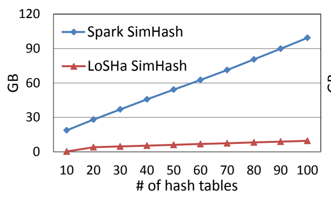


**(a) SimHash on** `Glove2.2M`

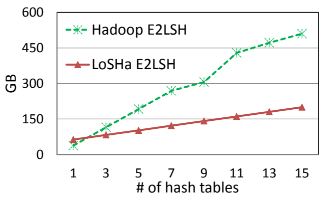


**(b) E2LSH on** `SIFT0.1B`

**Figure 3: Overall running time vs. Error ratio**



**(a) SimHash on** `Glove2.2M`



**(b) E2LSH on** `SIFT0.1B`

**Figure 4: Network traffic vs. Number of hash tables**

**Quality measure.** Given a query set $Q$, let $o_1, o_2, \ldots, o_K$ be the top $K$ ANNs returned by an LSH algorithm and $o_1^*, o_2^*, \ldots, o_K^*$ be the exact top $K$ NNs. Let $d(o, q)$ denote the distance between $o$ and $q$. *Error ratio*, $ER = (1/(|Q|K)) \sum_{q \in Q} \sum_{i=1}^{K} (d(o_i, q)/d(o_i^*, q))$, is widely used in existing work [7, 9, 20, 29, 31, 36] to measure the quality of their algorithm. A smaller error ratio means better quality of results, and an error ratio of 1 means that the exact results are found. But for single-probing LSH algorithms, there is no guarantee that $K$ answers will be found. Thus, for those algorithms, we only used the number of answers returned to calculate the error ratio, and removed the query from $Q$ if no answer was returned.

## 6.1 Comparison with LSH on Spark/Hadoop

We first compared LoSHa with specific LSH implementations on Spark and Hadoop. On Spark, SoundCloud implemented SimHash [3], which is used in production [28] and the most popular LSH implementation on Spark. We denote it by *Spark SimHash* and compared it with our implementation of SimHash on LoSHa, denoted by *LoSHa SimHash*. On Hadoop, we used the *Hadoop E2LSH* implementation provided by the authors of [1] and compared it with our implementation of E2LSH [5] on LoSHa, denoted by *LoSHa E2LSH*. Spark SimHash and Hadoop E2LSH have more than 800 and 1500 lines of Scala/Java code, respectively, while both LoSHa SimHash and LoSHa E2LSH have about 100 lines of C++ code.

Since both Spark SimHash and Hadoop E2LSH are slow and require a lot of computing resources to process large datasets, we could only test them on smaller datasets. SimHash is designed for angular distance and is used for querying text data, and thus we used the `Glove2.2M` dataset. We set $k = 20$ as normally used by Spark SimHash, and varied $L$ from 10 to 100. E2LSH is for Euclidean space and thus we used it to query the SIFT image descriptor dataset. However, `SIFT1B` is too large for Hadoop E2LSH and thus we only used 0.1 billion items, denoted by `SIFT0.1B`. We set $k = 20$ and varied $L$ from 1 to 15 (note that the running time of Hadoop E2LSH increases dramatically for larger $L$).

We report the overall running time of the four implementations in Figure 3. As we increase $L$, i.e., the number of hash tables, the running time also increases but the error ratio decreases. However, the running time of LoSHa increases much slower than Spark SimHash and Hadoop E2LSH. LoSHa SimHash attains an error ratio of 1.11 in 22.6 seconds, while Spark SimHash requires 411.6 seconds. LoSHa E2LSH attains an error ratio of 1.26 in 145.3 seconds, while Hadoop E2LSH uses 2,005.9 seconds.

The superior performance of LoSHa over the Spark/Hadoop implementations is mainly brought by our LSH-specific system implementation such as fine-grained data access and fast joins enabled by the data layout. In contrast, the Spark/Hadoop implementations follow the simple logic that joins queries and items by signatures (as discussed in Section 2.2.2), which incurs high network communication cost. As shown in Figure 4, LoSHa has significantly less network traffic (i.e., the total volume of messages sent over the network) than Spark/Hadoop, which verify the effectiveness of our LSH-specific system implementation. Note that Spark or Hadoop implementations may be made more efficient, but it will require non-trivial programming efforts from users. A better way could be to implement LoSHa upon Spark and Haoop, so that users need not worry about LSH-specific optimization details. Implementing the LoSHa framework upon Spark/Haoop, however, is not the objective of our work, which we leave as future work.

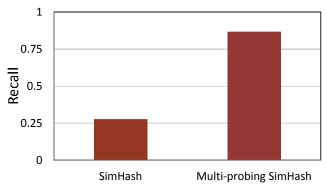
In Section 6.3, we will also compare LoSHa with the C++ MapReduce implementation as we examine in details the effectiveness of the optimization techniques in LoSHa.

## 6.2 Comparison with PLSH

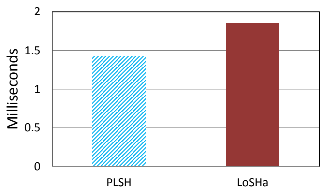
This experiment compared LoSHa with PLSH [30], a highly optimized distributed implementation of SimHash [3]. Unfortunately, we could not obtain the code of PLSH to repeat their experiments. We could only try to mimic their experiments on LoSHa. We summarize our results in Figure 5.

PLSH achieved a recall of 92% by building 780 hash tables for processing 1 billion tweets, which requires 6.4TB RAM in 100 machines. If we use SimHash, we can generate at most 55 hash tables with the total 0.96TB of memory in our cluster, thus giving a poor recall of 27.4% only. Thus, we implemented the multi-probing version of SimHash (denoted as multi-probing SimHash) using the "Candidate expansion using similar items" strategy in Section 3.3, which improves the recall from 27.4% to 86.8% with only 55 hash tables. We further measured the average querying time of multi-probing SimHash, which is 1.86ms (vs. 1.42ms of PLSH). The result is significant since LoSHa can process the same-scale data and achieve performance not much worse than PLSH which uses significantly more computing resources (i.e., RAM and CPU).

We emphasize that PLSH is a specific system highly optimized for SimHash only, while LoSHa is a general framework for implementing different types of LSH algorithms and thus it is expected that LoSHa has poorer performance than PLSH for SimHash. In fact, it is the multi-probing strategy that helps LoSHa to achieve high performance using much less resources than PLSH. Therefore, it is necessary for a system to be user-friendly for implementing multi-probing algorithms. Note that our implementation of multi-probing SimHash has about 100 lines of effective code only, but it is definitely non-trivial to implement PLSH [30]. Thus, we believe that



**(a) Recall**



**(b) Average querying time**

**Figure 5: Performance of multi-probing LoSHa**

**Table 2: Effectiveness of optimizations in LoSHa**

| | `Glove2.2M` | | `Tweets0.21B` | |
|---|---|---|---|---|
| | Querying time | Network traffic | Querying time | Network traffic |
| LoSHa | 0.52s | 0.140GB | 30.21s | 20.13GB |
| No de-duplication | 0.48s | 0.143GB | 82.38s | 74.98GB |
| No message reduction | 3.09s | 4.95GB | 282.29s | 526.95GB |
| MapReduce LoSHa | 187.61s | 355.34GB | failed | failed |

the abstraction, API and LSH-specific implementation of LoSHa can help promote multi-probing LSH on existing distributed platforms.

## 6.3 Effects of Optimization Techniques

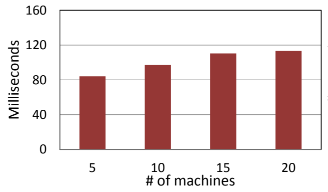
We now examine the effectiveness of the various optimizations implemented in LoSHa. To do that, we created four versions of LoSHa: (1) LoSHa with all optimizations enabled, (2) LoSHa without performing de-duplication, (3) LoSHa by disabling the message reduction techniques (i.e., query broadcasting and various combiners), and (4) LoSHa by applying MapReduce (in C++) instead of the two-level join.

We first tested the performance of the four versions of LoSHa running SimHash on `Glove2.2M`, with $k = 20$ and $L = 100$. We also tested on a larger dataset, `Tweets0.21B`, by processing 0.21 billion tweets from `Tweets1B`, i.e., each machine processes 10.5 million tweets. As `Tweets0.21B` is too large for single-probing SimHash to work, we used the multi-probing SimHash, with $k = 16$ and $L = 55$. We report the time and network traffic for processing 1,000 queries in Table 2. We discuss the effects of different optimizations as follows.

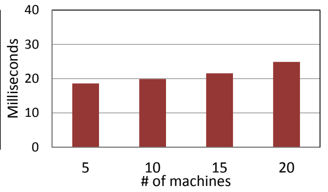
**Effect of de-duplication.** By disabling de-duplication, the querying time of LoSHa for `Glove2.2M` is even reduced from 0.52s to 0.48s. This is because word vectors in `Glove2.2M` are relatively different from each other and thus the amount of duplicate processing is small, as verified by the similar volumes of network traffic measured for LoSHa and LoSHa without de-duplication. Note that duplicate copies may also be already removed by the two-level combiner in LoSHa (see details in Sections 5.4 and 5.5). On the other hand, de-duplication itself incurs overhead and thus the overall time may be even reduced without de-duplication. For the tweets dataset, however, disabling de-duplication significantly degrades the performance and query processing is almost 3 times slower. This is because many tweets are highly similar and thus the amount of duplicate processing is much larger. In addition, we use multi-probing for `Tweets0.21B`, and hence duplicate candidates make queries probe more buckets in subsequent iterations. Thus, the volume of network traffic is also significantly increased from 20.13GB to 74.98GB when de-duplication is disabled in LoSHa.

**Effect of message reduction.** The importance of the message reduction techniques is obvious as disabling them immediately increases the network traffic by 35 times for `Glove2.2M` and 26 times for `Tweets0.21B`. As a result, communication cost dominates the total querying cost and thus the overall query performance is also severely degraded.

**Effect of two-level join.** LoSHa's data layout, which enables the two-level join for query processing, is the most critical optimization. When we replace the two-level join with a Map-Reduce procedure, a large number of messages are transmitted over the network and the processing time is also severely increased. Note that both machine-level combiner and de-duplication are already enabled. For `Tweets0.21B`, the system ran out of the (48 x 20)GB of aggregate memory in the cluster. Note that the Map-Reduce procedure is essentially a hash join between queries and items on their signatures, but it generates much more messages than the two-level join employed with LoSHa's data layout.



**(a) M-SimHash on** `Tweets1B`



**(b) M-E2LSH on** `SIFT1B`

**Figure 6: Average query time vs. Number of machines**

## 6.4 Scalability

We tested the scalability of LoSHa with the two billion-scale datasets, `Tweets1B` and `SIFT1B`. We varied the number of machines from 5 to 20, while fixing the number of data items per machine to 50 million.

For querying the tweets, we ran multi-probing SimHash (denoted by *M-SimHash*), with $k = 16$ and $L = 55$, as in Section 6.2. We report the average querying time for processing 1,000 queries in Figure 6a, which shows a gentle slowdown in query processing when the number of machines increases. Since the number of data items increases in the same rate as the increase in the number of machines, the small increase in querying time is reasonable as more data are transmitted through the network and also more coordination among workers is needed. We observed that there was no obvious straggler during the whole process and thus the slowdown is more likely to be caused by communication, for which a faster network (e.g., Infiniband instead of Gigabit Ethernet) will help reduce the slowdown.

We ran LoSHa E2LSH to query the SIFT data and used $k = 20$ as in Section 6.1. However, we could use at most 3 hash tables because the number of items is increased to 50 million per machine (note that unlike a tweet, a SIFT item is a dense vector). The error ratio is

1.72. Thus, we implemented multi-probing E2LSH on LoSHa, which reduces the error ratio to 1.23. There is also a small increase in querying time when we increase the number of machines from 5 to 20, as shown in Figure 6b, for the same reason as we explained for Figure 6a.

## 7 RELATED WORK

LSH [13] has been extensively studied and various hash function families for different metrics have been developed, such as $l_p$ distance [5], angular distance [3], Jaccard distance [2], and so on. Since single-probing [1, 18, 21, 22, 27, 28] requires a large number of hash tables and cannot scale to handle large datasets, many multi-probing LSH algorithms have been proposed to use fewer hash tables and yet achieve comparable quality of results [7, 12, 20, 24, 29, 31, 36]. However, they are single-core algorithms and many still rely on external memory to handle large datasets.

Recently more distributed LSH algorithms have been developed to handle larger datasets. PLSH [30] is a highly optimized LSH implementation for distributed SimHash [3], but the development cost of such a system tailor-made only for one specific LSH algorithm is too high for general applications. Most existing distributed LSH algorithms were implemented on general-purpose platforms such as Hadoop [1, 18, 21], Spark [22, 28], and distributed hash tables [11]. However, these general-purpose platforms are not designed for solving domain-specific problems and thus the LSH implementations on these platforms are also inefficient.

For solving problems in a particular domain, quite a number of domain-specific systems, such as Parameter Server [17], PowerGraph [10] and LFTF [35], etc., have been developed, with which one can easily implement different types of distributed machine learning and graph analytics algorithms in one framework. However, we are not aware of any work on the development of a general framework that allows efficient and scalable implementation of all kinds of LSH algorithms.

## 8 CONCLUSIONS

We presented LoSHa, which offers a general, unified programming framework and a user-friendly API for easy implementation of a wide range of LSH algorithms. We explored various LSH-specific system implementation and optimizations to enable scalable LSH computation. Our experiments on billion-scale datasets verified the efficiency, scalability, and quality of results obtained by LoSHa's implementations of a number of popular LSH algorithms. We believe that LoSHa can benefit many applications of LSH as it can significantly lower the development cost and achieve high performance. Such a general framework for LSH can also lead to the development of more efficient distributed algorithms for new LSH applications.

**Ack.** This work was partially supported by Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC, ITF 6904079, and Grant 3132821 funded by the Research Committee of CUHK.

## REFERENCES

[1] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *CIKM*, pages 2174–2178, 2012.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

[3] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[4] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[5] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[8] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. DSH: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, pages 1127–1138, 2014.

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[11] P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT*, pages 744–755, 2009.

[12] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. In *PVLDB*, volume 9, pages 1–12, 2015.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[14] Learning to Hash. http://cs.nju.edu.cn/lwj/l2h.html. 2017.

[15] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*, pages 2834–2842, 2014.

[16] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao. A comparison of general-purpose distributed systems for data processing. In *IEEE BigData*, pages 378–383, 2016.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[18] LikeLike. https://github.com/takahi-i/likelike. 2017.

[19] W. Liu, J. Wang, S. Kumar, and S. Chang. Hashing with graphs. In *ICML*, pages 1–8, 2011.

[20] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. SK-LSH: an efficient index structure for approximate nearest neighbor search. In *PVLDB*, volume 7, pages 745–756, 2014.

[21] LSH-Hadoop. https://github.com/lancenorskog/lsh-hadoop. 2017.

[22] LSH-Spark. https://github.com/marufaytekin/lsh-spark. 2017.

[23] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. In *PVLDB*, volume 8, pages 281–292, 2014.

[24] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[26] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. In *Pattern Recognition Letters*, volume 31, pages 1348–1358, 2010.

[27] A. Rajaraman, J. D. Ullman, J. D. Ullman, and J. D. Ullman. *Mining of massive datasets*, volume 1. 2012.

[28] SoundCloud-LSH. https://github.com/soundcloud/cosine-lsh-join-spark. 2017.

[29] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. In *PVLDB*, volume 8, pages 1–12, 2014.

[30] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. In *PVLDB*, volume 6, pages 1930–1941, 2013.

[31] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[32] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. In *CoRR*, volume abs/1408.2927, 2014.

[33] F. Yang, Y. Huang, Y. Zhao, J. Li, G. Jiang, and J. Cheng. The best of both worlds: Big data programming with both productivity and performance. In *SIGMOD*, pages 1619–1622, 2017.

[34] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. In *PVLDB*, volume 9, pages 420–431, 2016.

[35] F. Yang, F. Shang, Y. Huang, J. Cheng, J. Li, Y. Zhao, and R. Zhao. LFTF: A framework for efficient tensor analytics at scale. In *PVLDB*, volume 10, pages 745–756, 2017.

[36] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, 2016.