11-15-2013

# Parallel Similarity Join

Chi Zhang
*The Florida State University*

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

# PARALLEL SIMILARITY JOIN

By

CHI ZHANG

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 2013

Chi Zhang defended this dissertation on November 15, 2013.

The members of the supervisory committee were:

Xin Yuan
Professor Co-Directing Dissertation

Feifei Li
Professor Co-Directing Dissertation

Washington Mio
University Representative

Piyush Kumar
Committee Member

Zhenhai Duan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

# ACKNOWLEDGMENTS

This dissertation would not have been possible without the support of many individuals. First of all, I would like to thank my advisor, Prof. Feifei Li, for his tremendous support, patience and encouragement throughout my years at the Florida State University (FSU). His insightful comments and guidance at every stage of this dissertation were essential for me to complete my research and has taught me innumerable lessons and insights on the working of research in general. He serves not only as my academic advisor, but also as a lifetime friend.

I would like to thank my dissertation committee members, Prof. Xin Yuan (from the Department of Computer Science at FSU), Prof. Piyush Kumar (from the Department of Computer Science at FSU), Prof. Washington Mio (from the Department of Mathematics at FSU), and Prof. Zhenhai Duan (from the Department of Computer Science at FSU) for their insightful comments and advice. Especially, I would like to express my special thanks to Prof. Xin Yuan for his great guidance and help during my years at FSU.

I would also like to thank the members of the Database group at FSU for their friendships, encouragements, support and collaboration. The list includes Jeffrey Jestes, Wangchao Le, Mingwang Tang and Bin Yao.

I also want to express my great appreciations to Dr. Robert van Engelen, Dr. David Whalley, Dr. Michael Mascagni, and Mr Bob Myers in the Department of Computer Science for their supports. I want to show my special thanks to Ms. Kristan McAlpin, Ms. Eleanor McNealy, Mr. Daniel Clawson, Ms. Edwina Hall and Mr. Yu Wang for their constant kind assistance during my years in FSU. Also, I am very thankful to Dr. Jinfeng Zhang from the Department of Statistics for his great help and support.

Last but not least, I am deeply indebted to my family and I want to express my greatest gratitude to my family members, especially for my Dad and my Mom for their continuous understanding and love. Without their support, this dissertation would not be possible. Their love and support for me are always the most precious wealth in my life.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ABSTRACT

Similarity join has been widely studied and used in various scientific and commercial applications. Given two datasets, similarity join finds all pairs of similar objects (one from each input dataset), subject to a distance metric and a ranking threshold. Typical applications of similarity join operators include knowledge discovery, data mining tasks, and nearest neighbor search in spatial databases GIS systems. With the amount of data to be processed growing at an ever-increasing rate, efficient computation of similarity joins becomes a challenging and imperative task. Although researchers have devoted extensive efforts on this subject, a number of problems still remain open. In particular, efficiency and scalability pose significant challenges and have seriously limited the applications of similarity join operators. To that end, a tempting choice is to leverage the power of parallel computations from a cluster of commodity machines, e.g., the MapReduce computation framework that offers opportunities to handle large data sets with efficiency, scalability and ease of use.

Unfortunately, most existing work assumed a centralized, sequential setting, and cannot be applied to parallel and distributed systems. On the other hand, other existing work studied similarity joins in traditional parallel environments, such as in the MPI programming model, which are also not applicable in a shared-nothing cluster setting that MapReduce adopts. That said, focusing on the most popular type of similarity join operators, known as the $k$ nearest neighbor ($k$NN) join, this thesis has investigated the challenges of how to efficiently execute $k$NN joins both in low-dimensional and high-dimensional spaces for large data sets in MapReduce. We present solutions for both exact and approximate $k$NN joins. Our constructions for the approximate $k$NN joins are particularly interesting, where we trade accuracy for efficiency and scalability in a principal fashion.

# CHAPTER 1

# INTRODUCTION

Similarity search is a fundamental operation in numerous applications, in which the objective is to find one or more similar objects to an input object (also known as the query object) from a database, subject to a distance metric $d(\cdot)$. The most popular type of similarity search is known as the $k$ nearest neighbor ($k$NN) search, where the search returns the top $k$ objects that are most similar to the query object, i.e., the $k$ objects from the database with the smallest distances to the query object. This operation finds numerous applications in many different domains, and is a building block for a lot of analytical tasks in databases, data mining, machine learning, and many more [15, 27, 41]. Due to its importance, the efficient answering of $k$NN queries has been extensively studied in the literature, and various extensions based on $k$NN queries have been explored. On the other hand, the join operator is the most fundamental operation in a database. Not surprisingly, similarity join based on $k$NN queries quickly becomes an essential operation that finds many important applications.

In particular, the $k$-nearest neighbor join ($k$NN join) is frequently used in numerous applications including knowledge discovery, data mining, and spatial databases [11, 40, 53, 55]. Since both the join and the nearest neighbor (NN) search are expensive, especially on large data sets and/or in multi-dimensions, $k$NN-join is a costly operation. Extensive research efforts have been devoted to improve the performance of $k$NN joins by proposing efficient algorithms [53, 55]. However, all these approaches focus on methods that are to be executed in a single thread from a centralized environment. With the fast increase in the scale of the input datasets, processing large data in parallel and distributed fashions is becoming a popular practice. Even though a number of parallel algorithms for equi-joins in relational engines [29, 42], set similarity joins [47], relational $\theta$-joins [38], and spatial range joins defined by distance threshold [60] in MapReduce have been designed and implemented, there has been little work on parallel $k$NN joins in large data, which is a challenging task and becoming increasingly essential as data sets continue to grow at an exponential rate.

The *big data* promise also means that data sets often consist of multi-dimensional objects, for example, when dealing with image and document data, users usually extract a (large) number of features from these data and store the extracted features in a multi-dimensional database. It is not uncommon to have very high dimensionality in these application scenarios, for example, the popular SIFT framework extracts 128 features from any input image.

When dealing with extreme-scale data, parallel and distributed computing using shared-nothing clusters, which typically consist of a large number of commodity machines, is quickly becoming a dominating trend. MapReduce [20] was introduced with the goal of providing a simple yet powerful parallel and distributed computing paradigm. The MapReduce architecture is typically deployed in a cluster, potentially consisting of a huge number of shared-nothing (commodity) machines, which provides excellent scalability and fault tolerance mechanisms. A user of a MapReduce cluster only needs to define two functions, *map* and *reduce*, and by doing so transparently gains the parallel processing power of an entire cluster. In past years there has been increasing support for MapReduce from both industry and academia, making it one of the most actively utilized frameworks for parallel and distributed processing of large data today.

The MapReduce architecture is designed for dealing with extremely large amount of data sets and can scale up to tens of thousands machines. Therefore, it offers both opportunities and challenges to solve the *parallel similarity join* problems. Recently, [60] and [47] proposed algorithms to solve distance range join and set similarity join using MapReduce. Their work show the great potential to effectively executing $k$ nearest neighbor join in parallel using MapReduce.

Motivated by these observations, this thesis investigates the problem of executing $k$NN joins in parallel for large data using MapReduce. We examined challenges from both the low and high dimensional spaces, and studied both the exact and approximate solutions. By allowing approximations, we are able to trade-off accuracy for efficiency and scalability, but only doing so in a principal way where users are guaranteed with high quality approximations. To that end, we first propose the basic, exact approach using block-nested-loop-join (BNLJ), that works in any dimension, and its improved version using the R-tree indices in low dimensions. However, due to the quadratic number (to the number of blocks in each input dataset) of partitions produced (hence, the number of reducers), the basic approach does not scale well for large and/or high-dimensional data. In light of its limitation, we introduce a MapReduce-friendly, approximate algorithms.

To effectively answer approximate $k$NN joins in Euclidean space for low-dimensional data, we proposed a novel method, that is based on mapping multi-dimensional data sets into one dimension using space-filling curves ($z$-values), and transforming $k$NN joins into a sequence of one-dimensional range searches. We use a small number of random vectors to shift the datasets so that $z$-values can preserve the spatial locality with proved high probability. The most significant benefit of our approach is that it only requires a linear number (to the number of blocks in each input data set) of partitions (hence, the number of reducers), which features excellent scalability. There are a number of interesting and challenging problems associated with realizing this idea in MapReduce, e.g., how to perform the random shifts in MapReduce, how to design a good partition over the one-dimensional $z$-values for the join purpose, and instantiate it efficiently in MapReduce, how to reduce the amount of communication incurred in the Map-to-Reduce phase. We address these issues in our study.

Unfortunately, the above idea breaks down in high dimensions, since it is well known that space-filing curves become ineffective as dimensionality increases to very high dimensions (e.g., hundreds). Thus, for data in high dimensions, we introduce an efficient approximate $k$NN join algorithm that can run in parallel in MapReduce, by leveraging the locality sensitive hashing (LSH) method [22,27]. We also integrate this idea with the $z$-order based approximation after the data has been reduced into a lower dimension. There are many interesting challenges in realizing the above idea using MapReduce for high dimension data, such as how to reduce the variance in approximations, and how to deal with load balancing issues when projecting data into low-dimension buckets in LSH, which we will address in this thesis.

We present the details of the above algorithms in MapReduce and also illustrate how we handle a number of system issues which arise in realizing these algorithms using Hadoop. Extensive experiments over large real data sets (up to 160 million joining 160 million records and up to hundreds of dimensions) demonstrate that our approximate method consistently outperforms the basic approach by at least a few orders of magnitude, while achieving very good approximation quality.

The rest of this thesis is organized as follows. In Chapter 2, we survey the related work. In Chapter 3, we investigate the parallel $k$NN joins in low dimensions. Then, we extend our study to parallel $k$NN joins for high dimensions in Chapter 4. We conclude the dissertation and discuss interesting open problems and possible future work in Chapter 5,

# CHAPTER 2

# PRELIMINARY AND RELATED WORK

## 2.1 Definition of $k$ Nearest Neighbor Join

Formally, given two datasets $R$ and $S$ in $\mathbb{R}^d$. Each record $r \in R$ ($s \in S$) may be interpreted as a $d$-dimensional point. We focus on the $L_2$ norm, i.e., the similarity distance between any two records is their euclidean distance $d(r, s)$. Then, $\mathrm{knn}(r, S)$ returns the set of $k$ nearest neighbors ($k$NN) of $r$ from $S$, where ties are broken arbitrarily. Note that the nearest neighbor simply is the object from the database that has the smallest similarity distance to the query object. Many applications are also often interested in finding just the approximate $k$ nearest neighbors. Let $\mathrm{aknn}(r, S)$ be a set of approximate $k$ nearest neighbors for $r$ in $S$. Assume $r$'s exact $k$th nearest neighbor in $\mathrm{knn}(r, S)$ is point $p$. Let $p'$ be the $k$th nearest neighbor in $\mathrm{aknn}(r, S)$. Then, we say $\mathrm{aknn}(r, S)$ is a *$c$-approximation* of $\mathrm{knn}(r, S)$ for some constant $c$ if and only if:

$$d(r, p) \leq d(r, p') \leq c \cdot d(r, p).$$

The algorithm producing $\mathrm{aknn}(r, S)$ is dubbed a $c$-approximate $k$NN algorithm.

**$k$NN join**: The $k$NN join between $R$ and $S$, denoted as $\mathrm{knnJ}(R, S)$, is defined as:

$$\mathrm{knnJ}(R, S) = \{(r, \mathrm{knn}(r, S))| \text{ for all } r \in R\}.$$

In other word, the $k$NN join between $R$ and $S$ finds a pair for each record $r \in R$, where the first element of the pair is $r$ itself, and the second element of the pair is $r'$ $k$ nearest neighbors from $S$.

**Approximate $k$NN join**: The approximate $k$NN join of $R$ and $S$ is denoted as aknnJ$(R, S)$ and is expressed as:

$$\text{aknnJ}(R, S) = \{(r, \text{aknn}(r, S)) | \text{ for all } r \in R\}.$$

This definition is similar to the above, except that the second element in each pair in the output is a list of approximate $k$ nearest neighbors.

## 2.2 $k$NN Join for Centralized Environment

The $k$NN join operation is closely related with the range and the $k$ nearest neighbor ($k$NN) search; many $k$NN join algorithms were indeed extended from or evolved from range and $k$NN search algorithms. In what follows, we give a brief introduction to techniques and indexing structures used in range, $k$NN, and $k$NN join problems, assume the traditional, centralized and single-thread setting.

### 2.2.1 Exact $k$NN Search and $k$NN Join

**Exact $k$NN search.** For (relatively-low) multi-dimensional data, the $R$-tree [23] index and its variant $R^*$-tree [6] index provide very efficient way to perform range queries and $k$NN search in the Euclidean space. R-tree can be viewed as an extension to the $B^+$-tree to multi-dimensions. The main idea of R-tree is to group nearby points other and represent them with their minimum bounding rectangles(MBRs), which are recursively group into MBRs in the next higher level of the tree. $R^*$-tree index improves the performance of $R$-tree index by minimizing the area covered by a directory rectangle, the overlap between directory rectangles, and the margin of a directory rectangle. We illustrate a simple example of an

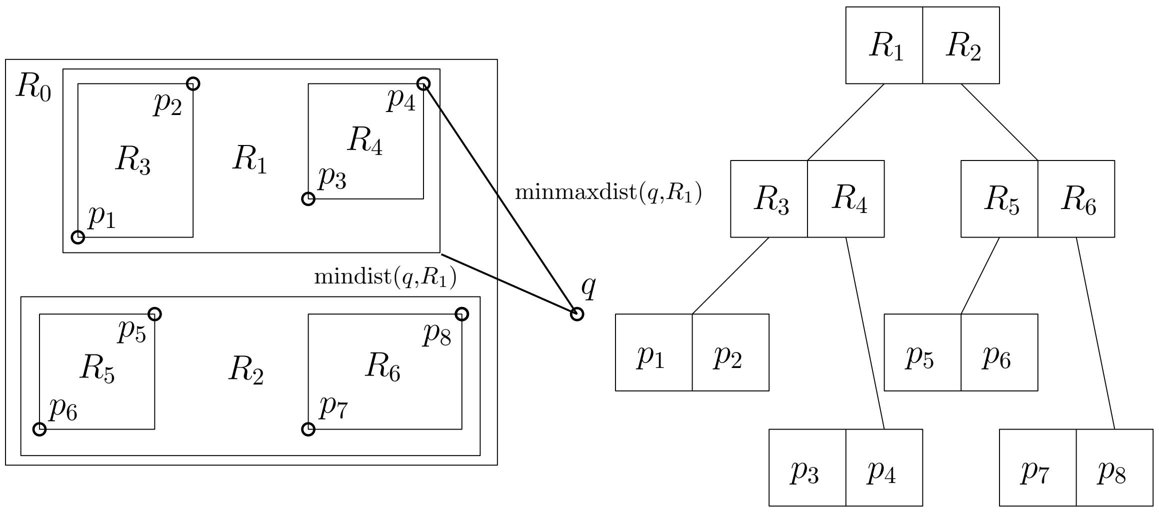


Figure 2.1: The R-tree.

R-tree index in Figure 2.1. Although R-tree cannot guarantee good worst-case performance, it generally has a good performance with real data in average cases for range and nearest neighbor queries.

Efficient R-tree based Range and $k$NN algorithms were introduced in [41] and [25]. The key idea of both algorithms is to adopt branch-and-bound search techniques. In particular, to answer $k$NN queries, the algorithm traverses an $R$-tree ether in the depth-first or the best-first manner, computes the distances (according to one of several metrics) between a query point $q$ and given MBRs, and order these MBRs for pruning the search tree. Such metrics include the mindist and the minmaxdist. The mindist is the minimum possible distance between a query point $q$ and an MBR $R$, such that there is no child rectangle or point inside $R$ can have a smaller distance. The minmaxdist measures the lower bound on the maximum distance for a point $q$ to any point in an MBR $R$. Figure 2.1 illustrates an example of these metrics. Given a query point $q$ and a nearest neighbor list (KNL) that is a list of candidate $k$NNs, we maintain a global maxdist (GM), which denotes the largest distance from $q$ to any point in KNL, and set GM to $+\infty$ if KNL contains less than $k$ members. We start $k$NN search from R-tree's root node, if current node is a leaf node, we simply compute the distances between the query point and every point in the node and update KNL and GM if necessary. Otherwise, we add all of its children nodes to an active branch list (ABL) and sort the ABL by it distance to $q$ in an ascending order (using mindists of points in ABL). Then, we could iterate through the sorted ABL and recursively invoke search on child nodes whose mindists less than GM and ignore any child node with a distance larger than GM. As we can see, the intuition behind using the mindist metric to prune the search space for $k$NN search is rather straightforward, e.g., when an MBR's mindist to $q$ is larger than the distance between the $k$th nearest neighbor and $q$, we can safely prune the entire MBR. When ABL becomes empty, the search can safely terminate.

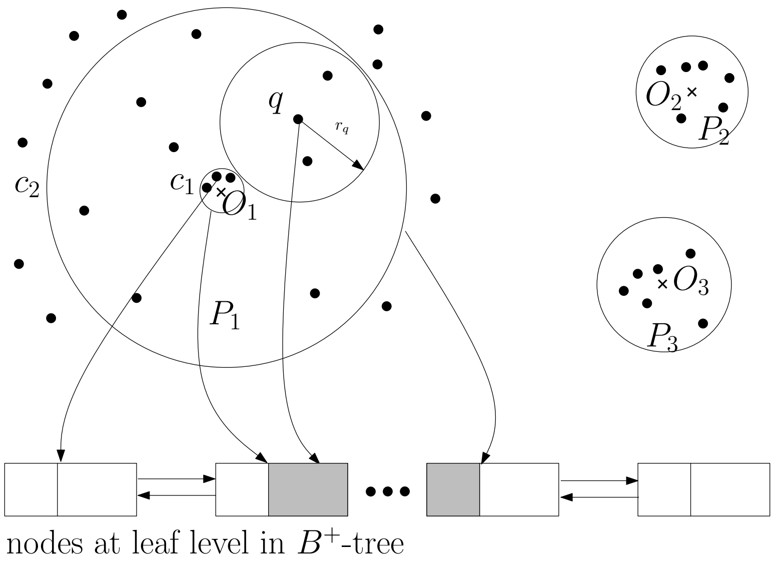


Figure 2.2: The iDistance index

In higher dimensions, e.g, $d \geq 6$, R-tree and its variant become ineffective. To process exact $k$NN queries efficiently for high-dimensional data sets, the iDistance method was introduced in [56]. The key idea of iDistance is to convert data in high-dimensional space into one-dimensional values using reference points, thus, we can reuse the existing one-dimensional indices such as $B^+$-tree to efficiently answer range queries. The intuition of iDistance can be described as follows. First, We could derive the similarity between two points from their distances to a selected point (referred to as reference point). Then, the points close to each other in high-dimensional space are expected to have similar distance to the same reference points. Note also that distance is single, one-dimensional value. To build an iDistance index on a data set $R$, we need to partition $R$ into clusters using any popular clustering method. Next, a reference point is decided for each partition; typically the cluster center is chosen as the reference point for each partition.

In the last step, all points are represented as single values based on their distances to their corresponding reference points. The $k$NN search algorithm using iDistance starts with searching a small query radius and gradually increases the radius until all $k$NNs are found. Since the search algorithm is eventually implemented on an $B^+$-tree, it is very efficient. Figure 2.2 gives an example of the iDistance index. In the example, data set $R$ has 3 partitions $P_1$, $P_2$, and $P_3$ with reference points $O_1$, $O_2$, and $O_3$, $r_q$ is the radius of query ball and we just need to check points fall in the circular belt region between $c1$ and $c2$, which corresponds the gray range in the $B^+$-tree.

In even higher dimensions, e.g., $d \geq 50$, iDistance and other similar nearest neighbor search algorithms (by mapping multi-dimensional data to a carefully constructed one-dimensional space) become ineffective. The best exact NN search algorithm is to simply scan the entire dataset.

**Exact $k$NN join.** Leveraging the rich results on exact $k$NN search, several exact $k$NN join algorithms have been proposed in the literature, for data sets in low or relatively high dimensions, which we will review next.

In [10, 11], a $k$NN join algorithm based on the Multipage Index (MuX) [12] inspired by the concept of the Minkowski sum [9] is proposed. It is essentially an R-tree based approach but is designed for solving the optimization conflict between CPU and I/O costs. The MuX index structure employs large pages (hosting pages) to optimize I/O cost and these large pages contain a secondary search structure to optimize CPU cost. The secondary search structure consists of smaller data buckets or directory buckets, which are optimized for CPU

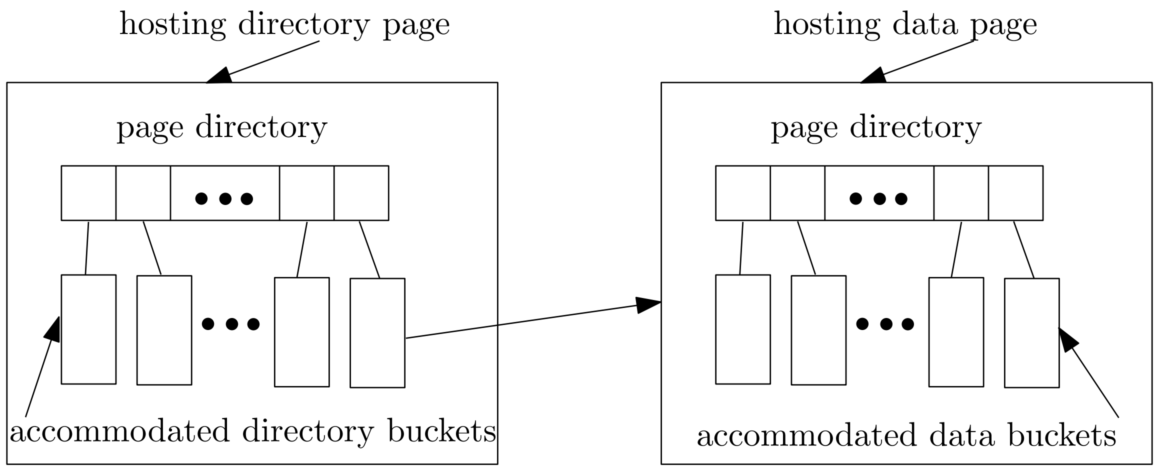


Figure 2.3: MuX index architecture

operations. Figure 2.3 illustrates example of the index architecture of MuX. The MuX $k$NN join algorithm works as follows. Given two data sets $R$ and $S$, it builds Mux index on both data sets. Then, it iterates over pages of $R$ and loads them into memory. Next, for every $R$ page it retrieves potential $S$ pages through the MuX index on $S$ and looking for $k$NN in $S$ pages.

The MuX index has better performance than R-tree index because of its separate optimizations of CPU and I/O costs. However, it shares the same drawbacks of R-tree index. As the dimensionality grows, the performance of deteriorates very quickly and eventually is equivalent to a linear scan algorithm for high-dimensional data to find $k$NN for a single query point. Moreover, the memory consumption of MuX index on high-dimensional, large data would be very high, which seriously limits its scalability.

Another exact $k$NN join algorithm for multi-dimensional data was introduced in [53]. The algorithm is a block nested loop join method but is enhanced using a number of techniques such as sorting, join scheduling and distance computation reduction and filtering to minimize I/O and CPU overheads. Wang at el. [49] proposed an inverted index-based $k$NN join algorithms for high-dimensional sparse dataset. The $k$NN search method using iDistance can be easily extended to handle $k$NN joins and an iDistance based exact $k$NN join algorithm for high-dimensional data was proposed in [55].

### 2.2.2 Approximate $k$NN Search and $k$NN Join

**Approximate $k$NN Search.** In many applications, users are satisfied by finding approximate $k$NNs, when an approximate solution can offer answers that are almost as good as the exact solutions, but delivers better efficiency and/or scalability. Furthermore, as we reviewed above, computing exact solutions is extremely expensive in very high dimensions, when any exact method simply degrades to the naive method of linearly scanning the whole data set. In these scenarios, approximate solutions that tradeoff the quality of the answer with the query efficiency become extremely attractive.

That said, for fixed dimensions (i.e., viewing the dimensionality $d$ as a constant), Arya et al. presents the Balanced Box Decomposition (BBD) tree in [5], which is designed based on a modified version of the standard kd-tree. Using an BBD tree, we can find an $(1+\epsilon)$-approximate nearest neighbor in $O(1/\epsilon^d \log N)$ time. BBD-tree takes $O(N \log N)$ time to build. However, the performance of BBD-tree degrades as the size of data set becomes huge (e.g. more than $100,000$) and/or dimensionality goes beyond 20. Recently, Bin et al. [54] introduces an $z$-order based approximate $k$NN search algorithm. The idea of the algorithm is to convert multi-dimensional points into one-dimensional value using $z$-order curves on multiple, randomly shifted copies of the input data set (to preserve the locality). Then, it builds $B^+$-trees to effectively answer $k$NN queries, by executing a number of one dimensional range queries.

For data sets in very high dimensions, most applications resort to the locality sensitive hashing (LSH) [22] technique to answer approximate $k$NN queries. The LSH-based technique essentially projects data from a high dimension space to a lower dimension, using random projections achieved by a family of LSH functions. An LSH function has nice properties that nearby points have the same hash value with high probability and two faraway points would likely to have different hash values. Ta et al. [46] presents the LSB-tree for approximate $k$NN retrieval in high-dimensional space, that integrates the ideas from both

LSH techniques and space-filling curves. In particular, an LSB-tree can be constructed as follows. Given a $d$-dimensional data set $R$, we use LSH functions to transform every point $o$ of $R$ into an $m$-dimensional point $M(o)$, then we convert $M(o)$ into a single-dimensional value, using the $z$-order value of $M(o)$. Next, we create $B^+$-tree index on these $z$-order values, thus in the end an LSB-tree consists of a conventional $B^+$-tree index over the $z$-order values of the random projections produced by LSH functions. To ensure better theoretical guarantee of the $k$NN query results and reduce variance of the approximations, we can build multiple LSB-trees referred to as an LSB-forest. To answer a $k$NN query, it is sufficient to do a range queries (of roughly size $k$) on every single LSB-tree from the LSB-forest using the $z$-order value of a query point as the center of the query range. The construction of an LSB-tree consumes $O((dn/B)^{1.5})$ space. And the LSB-trees based algorithm returns a 4-approximate NN with at least constant probability, in $O(E\sqrt{dn/B})$ IOs, where $E$ is the height of an LSB-tree.

**Approximate $k$NN Join.** Little has been done in the literature on producing approximate $k$NN join results. A straightforward solution is to simply take one of the approximate $k$NN search algorithms that we have reviewed above, and apply that on every point from $R$ over the second data set $S$.

## 2.3 Parallel $k$NN Joins

Compared with extensive research studies for $k$NN joins in the traditional, centralized and single-thread settings, there are very few existing work conducted on parallel $k$NN joins. Early research on parallel join algorithms, *for relational join operators*, in a shared-nothing multiprocessor environment has been presented in [29, 42]. However, since they dealt with relational join operators (such as equi-join or $\theta$-join), they are not applicable for solving the parallel $k$NN join problem.

Parallel spatial join algorithms using a multiprocessor system have been studied in [13,26,35,37,39,61], where the goal is to join two spatial datasets using to a spatial predicate (e.g., the intersection between two objects, specifying a threshold value on the distance between two objects). None of which is designed for efficiently processing $k$NN-joins over two datasets. Recently, Zhang et al. [59, 60] proposed a parallel spatial join algorithm in MapReduce, however, dealing with only spatial distance joins, which again does not solve the $k$NN-join problem.

Another closely related problem to $k$NN join is to produce a $k$NN graph. The $k$NN graph problem is a special instance of the general $k$NN join problem, in which it executes a self-join, i.e., there is only one input data set $R$ and the goal is to execute a $k$NN join between two (identical) copies of $R$. That said, in a $k$NN graph, each point in $R$ is connected to its $k$ nearest neighbors from the same dataset $R$. Efficient parallel computation of the $k$NN graph over a dataset using the Message Passing Interface (MPI) was proposed in [40]. Dilin et al. [48] proposed a parallel approximate $k$NN graph construction algorithm using a mixed code model of MPI and OpenMP [3]. Another parallel $k$NN graph construction method appears in [17, 18], which works for constructing the $k$NN graph over a centralized dataset. The goal there is to minimize cache misses for multi-core machines. Finally, an approximate $k$NN graph algorithm for generic similarity measures using MapReduce was proposed in [21]. Nevertheless, these parallel $k$NN graphs do not generalize to solve the general $k$NN join problem efficiently; since the latter needs to deal with two different input data sets.

The most closely related work appeared recently in [34]. In which, a parallel exact $k$NN join algorithm using MapReduce was proposed. The key idea of the algorithm is to divide the data sets into groups using Voronoi diagram-base partitioning policy and compute the $k$NN join by examining point pairs contained in the same group. However, it cannot handle large datasets efficiently and is not designed for processing high-dimensional data sets, as we will demonstrate in this thesis.

# CHAPTER 3

# PARALLEL $K$NN JOIN IN LOW DIMENSIONS

## 3.1 Motivation

Performing $k$NN joins in the traditional setup has been extensively studied in the literature [11, 53–55, 57]. Nevertheless, these works focus on the centralized, single-thread setting that is not directly applicable in parallel and distributed settings. Most previous research works either focus on relational join operator such as in [29, 42] or spatial intersection join or spatial distance join as in [13, 26, 35, 37, 39, 59–61]. Various parallel $k$NN graph algorithms have been presented in [17, 21, 40, 48], however, $k$NN graph only over the same data set instead of performing two data sets. [34] proposed a Voronoi diagram based $k$NN join algorithm using MapReduce, but it cannot handle very large dataset. Other close studies related to $k$NN join appear in [4, 45] where $k$NN queries were examined in MapReduce, by leveraging on the voronoi diagram and the locality-sensitive hashing (LSH) method, respectively. Their focus is the single query processing, rather than performing the joins. Given a $k$NN query algorithm, it is still quite challenging to design efficient MapReduce based $k$NN join algorithms (i.e., how to design partitioning to minimize communication, how to achieve load balancing). Solutions proposed in [50, 52] require pre-processing one dataset offline (not using MapReduce) and organize them into an overlay structure to build a global index that might be then adapted into a MapReduce cluster or possible solutions built on systems/frameworks that extend the standard MapReduce paradigm [14]. Join processing in MapReduce leveraging columnar storage was investigated in [32], which does require a new file format.

Our goal is to design practical, efficient $k$NN join algorithms that work well from 2 up to tens of dimensions (say 30, as shown in our experiments). Finally, our focus in this work is to work with ad-hoc join requests over ad-hoc datasets (both $R$ and $S$) using the standard MapReduce system and only standard *map* and *reduce* programming model (where it is best to avoid using sophisticated data structures and indices), hence, we do not consider solutions leveraging the global indexing structures.

## 3.2 MapReduce and Hadoop Basics

A MapReduce program typically consists of a pair of user-defined *map* and *reduce* functions. The map function is invoked for every record in the input data sets and produces a partitioned and sorted set of intermediate results. The reduce function fetches sorted data, from the appropriate partition, produced by the map function and produces the final output data. Conceptually, they are: $map(k1, v1) \rightarrow list(k2, v2)$ and $reduce(k2, list(v2)) \rightarrow list(k3, v3)$.

A typical MapReduce cluster consists of many slave machines, which are responsible for performing map and reduce tasks, and a single master machine which oversees the execution of a MapReduce program. A file in a MapReduce cluster is usually stored in a distributed file system (DFS), which splits a file into equal sized chunks (aka splits). A file's splits are then distributed, and possibly replicated, to different machines in the cluster. To execute a MapReduce job, a user specifies the input file, the number of desired map tasks $m$ and reduce tasks $r$, and supplies the *map* and *reduce* function. For most applications, $m$ is the same as the number of splits for the given input file(s). The coordinator on the master machine creates a map task for each of the $m$ splits and attempts to assign the map task to a slave machine containing a copy of its designated input split. Each map task partitions its output into $r$ buckets, and each bucket is sorted based on a user-defined comparator on

$k2$. Partitioning is done via a hash function on the key value, i.e. $hash(k2) \mod r$. A map task's output may be temporarily materialized on a local file system (LFS), which is removed after the map task completes.

Next, the coordinator assigns $r$ reduce tasks to different machines. A shuffle and sort stage is commenced, during which the $i$'th reducer $r_i$ copies records from $b_{i,j}$, the $i$'th bucket from each of the $j$th ($1 \leq j \leq m$) map task. After copying all $list(k2, v2)$ from each $b_{i,j}$, a reduce task merges and sorts them using the user specified comparator on $k2$. It then invokes the user specified reduce function. Typically, the reduce function is invoked once for each distinct $k2$ and it processes a $k2$'s associated list of values $list(v2)$, i.e. it is passed a $(k2, list(v2))$ pair per invocation. However, a user may specify a group-by comparator which specifies an alternative grouping by $k2$. For every invocation, the reduce function emits 0 or more final key value pairs $(k3, v3)$. The output of each reduce task $(list(k3, v3))$ is written into a separate distributed file residing in the DFS.

To reduce the network traffic caused by repetitions of the intermediate keys $k2$ produced by each mapper, an optional *combine* function for merging output in map stage, $combine(k2, list(v2)) \rightarrow list(k2, v2)$, can be specified.

It might be necessary to replicate some data to all slaves running tasks, i.e. job specific configurations. In Hadoop [24] this may be accomplished easily by submitting a file to the master for placement in the *Distributed Cache* for a job. All files submitted to a job's Distributed Cache are replicated to all slaves during the initialization phases of the job and removed after the completion of the job. We assume the availability of a Distributed Cache, though a similar mechanism should be available in most other MapReduce frameworks.

## 3.3 Baseline Methods

The straightforward method for $k$NN-joins in MapReduce is to adopt the block nested loop methodology. The basic idea is to partition $R$ and $S$, each into $n$ equal-sized blocks in the Map phase, which can be easily done in a linear scan of $R$ (or $S$) by putting every $|R|/n$

(or $|S|/n$) records into one block. Then, every possible pair of blocks (one from $R$ and one from $S$) is partitioned into a bucket at the end of the Map phase (so a total of $n^2$ buckets). Then, $r$ $(= n^2)$ reducers are invoked, one for each bucket produced by the mappers. Each reducer reads in a bucket and performs a block nested loop $k$NN join between the local $R$ and $S$ blocks in that bucket, i.e., find $k$NNs in the local block of $S$ of every record in the local block of $R$ using a nested loop. The results from all reducers are written into $(n^2)$ DFS files. We only store the records' ids, and the distance between every record $r \in R$ to each of its $k$NNs from a local $S$ block, i.e., the record output format of this phase is $(rid, sid, d(r, s))$.

Note in the above phase, each record $r \in R$ appears in one block of $R$ and it is replicated in $n$ buckets (one for each of the $n$ blocks from $S$). A reducer for each bucket simply finds the local $k$NNs of $r$ w.r.t. the corresponding block from $S$. Hence, in the second MapReduce phase, our job is to find the global $k$NNs for every record $r \in R$ among its $n$ local $k$NNs produced in the first phase, a total of $nk$ candidates. To do so, it is necessary to sort the triples (a record $r \in R$, one of its local $k$NNs in one bucket, their distance) from the first phase for each $r \in R$. This can be achieved as follows. In the Map stage, we read in all outputs from the first phase and use the unique record ID of every record $r \in R$ as the partitioning key (at the end of the Map phase). Hence, every reducer retrieves a $(rid, list(sid, d(r, s))$ pair and sorts the $list(sid, d(r, s))$ in ascending order of $d(r, s)$. The reducer then emits the top-$k$ results for each $rid$. We dub this method *H-BNLJ* (Hadoop Block Nested Loop Join).

**An improvement.** A simple improvement to *H-BNLJ* is to build an index for the local $S$ block in a bucket in the reducer, to help find $k$NNs of a record $r$ from the local $R$ block in the same bucket. Specifically, for each block $S_{bj}$ $(1 \leq j \leq n)$, we build a reducer-local spatial index over $S_{bj}$, in particular we used the R-tree, before proceeding to find the local $k$NNs for every record from the local $R$ block in the same bucket with $S_{bj}$. Then, we use $k$NN functionality from R-tree to answer $\text{knn}(r, S_{bj})$ in every bucket in the reducer. Bulk-loading a R-tree for $S_{bj}$ is very efficient, and $k$NN search in R-tree is also efficient, hence this overhead is compensated by savings from not running a local nested loop in each bucket. Remaining steps are identical to *H-BNLJ*, and we dub it *H-BRJ* (Hadoop Block R-tree Join).

**Cost analysis.** Each bucket has one local $R$ block and one local $S$ block. Given $n^2$ buckets to be read by the reducer, the communication cost of *H-BNLJ* in the first MapReduce round is $O((|R|/n + |S|/n) \cdot n^2)$. In the second round, for each record $r \in R$, all local $k$NNs of $r$ in each of the $n$ buckets that $r$ has been replicated into are communicated. So the communication cost of this round is $O(kn|R|)$. Hence, the total communication cost of *H-BNLJ* is $O(n(|R| + |S|) + nk|R|)$. In terms of cpu cost, the cost for each bucket in the first round is clearly $(|R||S|)/n^2$. Summing over all buckets, it leads to $O(|R||S|)$. In the second round, the cost is to find the $k$ records with the smallest distances among the $nk$ local $k$NNs for each record $r \in R$. Using a priority queue of size $k$, this can be achieved in $O(nk \log k)$ cost. Hence, the total cpu cost of *H-BNLJ* is $O(|R||S| + |R|nk \log k)$.

*H-BRJ*'s communication cost is identical to that in *H-BNLJ*. Bulk-loading an R-tree over each block $S_{bj}$ takes $O(|S_{bj}| \log |S_{bj}|)$ in practice. Note, there are $n^2$ buckets in total (each block $S_{bj}$ is replicated in $n$ buckets). The asymptotic worst-case query cost for $k$NN search in R-tree is as expensive as linear scan, however, in practice, this is often just square-root of the size of the dataset. Thus, finding local $k$NNs from $S_{bj}$ for each record in a local block $R_{bi}$ in a bucket with $R_{bi}$ and $S_{bj}$ takes $O(|R_{bi}|\sqrt{|S_{bj}|})$. Since $|R_{bi}| = |R|/n$ and $|S_{bj}| = |S|/n$ for any block, *H-BRJ*'s first round cpu cost is $O(n|S| \log(|S|/n) + n|R|\sqrt{|S|/n})$. Its second round is identical to that in *H-BNLJ*. Hence, *H-BRJ*'s total cpu cost is $O(n|S| \log(|S|/n) + n|R|\sqrt{|S|/n} + |R|nk \log k)$.

**Remarks.** In this work our goal is to work with ad-hoc join requests in a MapReduce cluster over datasets $R$ and $S$ which are stored in the cluster, and design algorithms that do everything in MapReduce. Therefore, we do not consider solutions which require preprocessing of the datasets either outside of the MapReduce cluster or using non MapReduce programs. We focus on using the standard MapReduce programming model, which always includes the *map* and *reduce* functions, to ensure high compatibility of our approaches with any MapReduce system. With these constraints, we do not consider the use of complex data structures or indices, such as a global (distributed) index or overlay structure built on a dataset offline, using non MapReduce programs, and adapted for later-use in the MapReduce cluster [50, 52].

## 3.4 Z-value Based Partition Join

In *H-BNLJ* and *H-BRJ*, for each $r \in R$ we must compute $\text{knn}(r, S_{bj})$ for a block $S_{bj}$ in a bucket, where each bucket is forwarded to a reducer for processing (leading to $n^2$ reducers). This creates excessive communication ($n^2$ buckets) and computation costs, as shown in our cost analysis.

This motivates us to find alternatives with linear communication and computation costs (to the number of blocks $n$ in each input dataset). To achieve this, we search for approximate solutions instead, leveraging on space-filling curves. Finding approximate nearest neighbors efficiently using space-filling curves is well studied, see [54] for using the $z$-order curve and references therein. The trade-off of this approach is spatial locality is not always preserved, leading to potential errors in the final answer. Fortunately, remedies exist to guarantee this happens infrequently:

**Theorem 1.** *[from [54]] Given a query point $q \in \mathbb{R}^d$, a data set $P \subset \mathbb{R}^d$, and a small constant $\alpha \in \mathbb{Z}^+$. We generate $(\alpha - 1)$ random vectors $\{\mathbf{v}_2, \ldots, \mathbf{v}_\alpha\}$, such that for any $i$, $\mathbf{v}_i \in \mathbb{R}^d$, and shift $P$ by these vectors to obtain $\{P_1, \ldots, P_\alpha\}$ ($P_1 = P$). Then, Algorithm 1 returns a constant approximation in any fixed dimension for* $\text{knn}(q, P)$ *in expectation.*

For any $p \in \mathbb{R}^d$, $z_p$ denotes $p$'s $z$-value and $\mathrm{Z}_P$ represents the sorted set of all $z$-values for a point set $P$; we define:

$$z^-(z_p, k, P) = k \text{ points immediately preceding } z_p \text{ in } \mathrm{Z}_P \tag{3.1}$$

**Algorithm 1**: zkNN($q$, $P$, $k$, $\alpha$)    [from [54]]

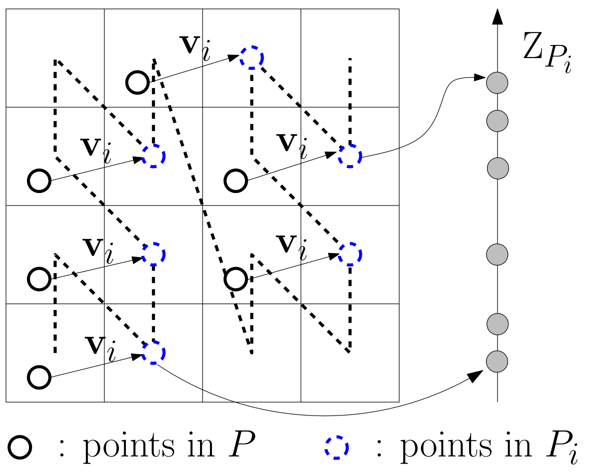
```
1 generate $\{\mathbf{v}_2, \ldots, \mathbf{v}_\alpha\}$, $\mathbf{v}_1 = \overrightarrow{0}$, $\mathbf{v}_i$ is a random vector in $\mathbb{R}^d$; $P_i = P + \mathbf{v}_i$ ($i \in [1, \alpha]$;
  $\forall p \in P$, insert $p + \mathbf{v}_i$ in $P_i$);
2 for $i = 1, \ldots, \alpha$ do
3     let $q_i = q + \mathbf{v}_i$, $C_i(q) = \emptyset$, and $z_{q_i}$ be $q_i$'s $z$-value;
4     insert $z^-(z_{q_i}, k, P_i)$ into $C_i(q)$;    // see (3.1)
5     insert $z^+(z_{q_i}, k, P_i)$ into $C_i(q)$;    // see (3.2)
6     for any $p \in C_i(q)$, update $p = p - \mathbf{v}_i$;
7 $C(q) = \bigcup_{i=1}^{\alpha} C_i(q) = C_1(q) \cup \cdots \cup C_\alpha(q)$;
8 return knn$(q, C(q))$.
```

$$z^+(z_p, k, P) = k \text{ points immediately succeeding } z_p \text{ in } \mathrm{Z}_P \tag{3.2}$$

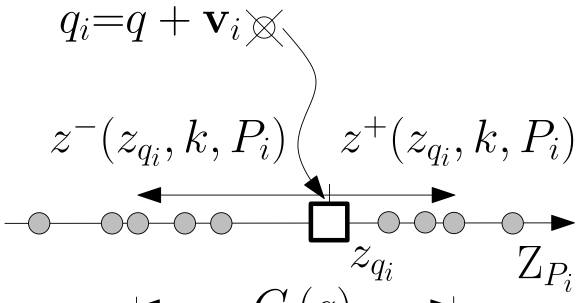
The idea behind zkNN is illustrated in Figure 3.1, which demonstrates finding $C_i(q)$ on $P_i$ ($= P + \mathbf{v}_i$) with $k = 3$. Algorithm zkNN repeats this process over $(\alpha - 2)$ randomly shifted copies, plus $P_1 = P$, to identify candidate points $\{C_1(q), \ldots, C_\alpha(q)\}$, and the overall candidate set $C(q) = \bigcup_{i=1}^{\alpha} C_i(q)$. As seen in Figure 3.1(b), $C_i(q)$ over $P_i$ can be efficiently found using binary search if $z$-values of points in $P_i$ are sorted, with a cost of $O(\log |P|)$. There is a special case for (3.1) (or (3.2)) near the head (tail) of $\mathrm{Z}_P$, when there are less than $k$ *preceding (succeeding)* points. When this happens, we take an appropriate additional number of *succeeding (preceding)* points, to make $k$ total points.

In practice, only a small number of random shifts are needed; $\alpha = 2$ is sufficient to guarantee high quality approximations as shown in our experiments.

Using zkNN for kNN joins in a centralized setting is obvious (applying zkNN over $S$ for every record in $R$, as Yao et al. did [54]). However, to efficiently apply the above idea for $k$NN joins in a MapReduce framework is not an easy task: we need to partition $R$ and $S$ delicately to achieve good load-balancing. Next, we will discuss how to achieve approximate $k$NN-joins in MapReduce by leveraging the idea in zkNN, dubbed *H-zkNNJ* (Hadoop based zkNN Join).



(a) $z$-values.



(b) Search on $z$-values.

Figure 3.1: Algorithm zkNN.

### 3.4.1 zkNN Join in MapReduce

**Overview of H-zkNNJ.** We generate $\alpha$ vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_\alpha\}$ in $\mathbb{R}^d$ (where $\mathbf{v}_1 = \overrightarrow{0}$) randomly, and shift $R$ and $S$ by these vectors to obtain $\alpha$ randomly shifted copies of $R$ and $S$ respectively, which are denoted as $\{R_1, \ldots, R_\alpha\}$ and $\{S_1, \ldots, S_\alpha\}$. Note that $R_1 = R$ and $S_1 = S$.

Consider any $i \in [1, \alpha]$, and $R_i$ and $S_i$, $\forall r \in R_i$, the candidate points from $S_i$, $C_i(r)$, for $r$'s $k$NN, according to zkNN, is in a small range ($2k$ points) surrounding $z_r$ in $Z_{S_i}$ (see Figure 3.1(b) and lines 4-5 in Algorithm 1). If we partition $R_i$ and $S_i$ into blocks along their $z$-value axis using *the same set* of $(n-1)$ $z$-values, $\{z_{i,1}, \ldots, z_{i,n-1}\}$, and denote resulting blocks as $\{R_{i,1}, \ldots, R_{i,n}\}$ and $\{S_{i,1}, \ldots, S_{i,n}\}$, such that:

$$R_{i,j}, S_{i,j} = [z_{i,j-1}, z_{i,j}) \text{ (let } z_{i,0} = 0,\ z_{i,n} = +\infty); \tag{3.3}$$

Then, for any $r \in R_{i,j}$, we can find $C_i(r)$ *using only* $S_{i,j}$, as long as $S_{i,j}$ contains at least $2k$ neighboring points of $z_r$, i.e., $z^-(z_r, k, S_i)$ and $z^+(z_r, k, S_i)$, as shown in Figure 3.2.

When this is not the case, i.e., $S_{i,j}$ does not contain enough ($k$) preceding (succeeding) points w.r.t. a point $r$ with $z_r \in R_{i,j}$, clearly, we can continue the search to the immediate left (right) block of $S_{i,j}$, i.e., $S_{i,j-1}$ ($S_{i,j+1}$), and repeat the same step if necessary until $k$

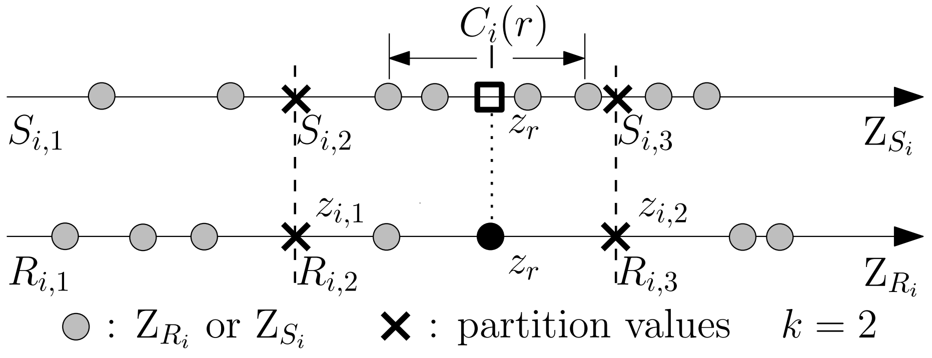


Figure 3.2: $C_i(r) \subseteq S_{i,j}$.

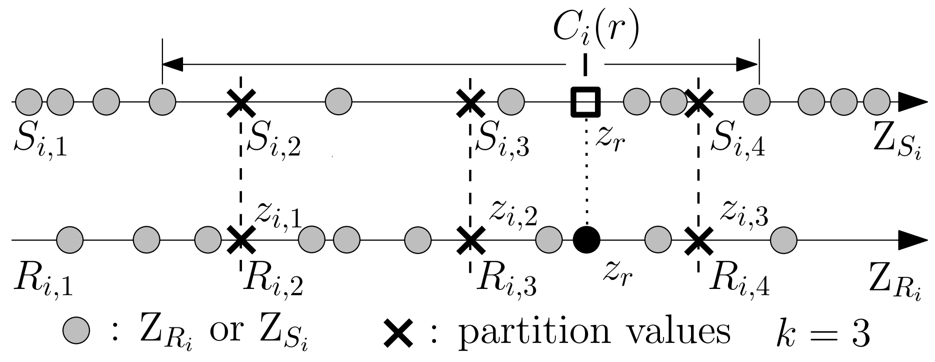


Figure 3.3: $C_i(r) \not\subseteq S_{i,j}$.

preceding (succeeding) points are met. An example is shown in Figure 3.3. To avoid checking multiple blocks in this process and ensure a search over $S_{i,j}$ is sufficient, we "duplicate" the nearest $k$ points from $S_{i,j}$'s preceding (succeeding) block (and further block(s) if necessary, when $S_{i,j-1}$ or $S_{i,j+1}$ does not have $k$ points). This guarantees, for any $r \in R_{i,j}$, the $k$ points with preceding (succeeding) $z$-values from $S_i$ are all contained in $S_{i,j}$, as shown in Lemma 1. This idea is illustrated in Figure 3.4.

**Lemma 1.** *By copying the nearest $k$ points to the left and right boundaries of $S_{i,j}$ in terms of $Z_{S_i}$, into $S_{i,j}$; for any $r \in R_{i,j}$, $C_i(r)$ can be found in $S_{i,j}$.*

*Proof.* By equations (3.1), (3.2), and (3.3), $z^-(z_{i,j-1}, k, S_i)$ and $z^+(z_{i,j}, k, S_i)$ are copied

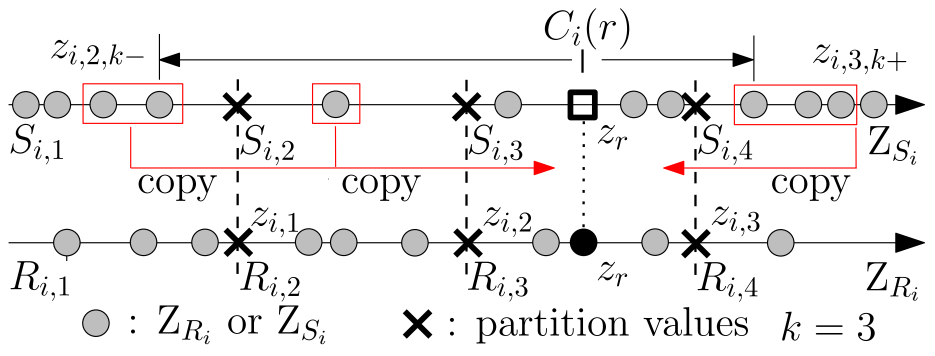


Figure 3.4: Copy $z^-(z_{i,j-1}, k, S_i), z^+(z_{i,j}, k, S_i)$ to $S_{i,j}$.

into $S_{i,j}$, i,e.,

$$S_{i,j} = [z_{i,j-1}, z_{i,j}) \cup z^-(z_{i,j-1}, k, S_i) \cup z^+(z_{i,j}, k, S_i). \tag{3.4}$$

Also, for any $r \in R_i$, $C_i(r) = z^-(z_r, k, S_i) \cup z^+(z_r, k, S_i)$. Since $r \in R_{i,j}$, we must have $z_{i,j-1} \leq z_r < z_{i,j}$. These facts, together with (3.4), clearly ensure that $C_i(r) \subseteq S_{i,j}$. □

That said, we can efficiently identify $C_i(r)$ for any $r \in R_{i,j}$, by searching only block $S_{i,j}$.

**Partition.** The key issue left is what partition values, as $\{z_{i,1}, \ldots, z_{i,n-1}\}$, delivers good efficiency in a distributed and parallel computation environment like MapReduce.

We first point out a constraint in our partitioning; the $j$th blocks from $R_i$ and $S_i$ must share the same boundaries. This constraint is imposed to appeal to the distributed and parallel computation model used by MapReduce. As discussed in Section 3.2, each reduce task in MapReduce is responsible for fetching and processing one partition created by each map task. To avoid excessive communication and incurring too many reduce tasks, we must ensure for a record $r \in R_{i,j}$, a reduce task knows where to find $C_i(r)$ and can find $C_i(r)$ locally (from the partition, created by mappers, which this reducer is responsible for). By imposing the same boundaries for corresponding blocks in $R_i$ and $S_i$, the search for $C_i(r)$ for any $r \in R_{i,j}$ should start in $S_{i,j}$, by definition of $C_i(r)$. Our design choice above ensures $C_i(r)$ is identifiable by examining $S_{i,j}$ only. Hence, sending the $j$th blocks of $R_i$ and $S_i$ into one reduce task is sufficient. This means only $n$ reduce tasks are needed (for $n$ blocks on $R_i$ and $S_i$ respectively). This partition policy is easy to implement by a map task once the partition values are set. What remains to explain is how to choose these values.

As explained above, the $j$th blocks $R_{i,j}$ and $S_{i,j}$ will be fetched and processed by one reducer and there will be a total of $n$ reducers. Hence, the best scenario is to have $\forall j_1 \neq j_2$, $|R_{i,j_1}| = |R_{i,j_2}|$ and $|S_{i,j_1}| = |S_{i,j_2}|$, which leads to the optimal load balancing in MapReduce. Clearly, this is impossible for general datasets $R$ and $S$. A compromised choice is to ensure $\forall j_1 \neq j_2$, $|R_{i,j_1}| \cdot |S_{i,j_1}| = |R_{i,j_2}| \cdot |S_{i,j_2}|$, which is also impossible given our partition policy. Among the rest (with our partitioning constraint in mind), two simple and good choices are to ensure the: (1) same size for $R_i$'s blocks; (2) same size for $S_j$'s blocks.

In the first choice, each reduce task has a block $R_{i,j}$ that satisfies $|R_{i,j}| = |R_i|/n$; the size of $S_{i,j}$ may vary. The worst case happens when there is a block $S_{i,j}$ such that $|S_{i,j}| = |S_i|$, i.e., all points in $S_i$ were contained in the $j$th block $S_{i,j}$. In this case, the $j$th reducer does most of the job, and the other $(n-1)$ reducers have a very light load (since for any other block of $S_i$, it contains only $2k$ records, see (3.4)). The bottleneck of the Reduce phase is apparently the $j$th reduce task, with the cost of $O(|R_{i,j}| \log |S_{i,j}|) = O(\frac{|R_i|}{n} \log |S_i|)$.

In the second choice, each reduce task has a block $S_{i,j}$ that satisfies $|S_{i,j}| = |S_i|/n$; the size of $R_{i,j}$ may vary. The worst case happens when there is a block $R_{i,j}$ such that $|R_{i,j}| = |R_i|$, i.e., all points in $R_i$ were contained in the $j$th block $R_{i,j}$. In this case, the $j$th reducer does most of the job, and the other $(n-1)$ reducers have effectively no computation cost. The bottleneck of the Reduce phase is the $j$th reduce task, with a cost of $O(|R_{i,j}| \log |S_{i,j}|) = O(|R_i| \log \frac{|S_i|}{n}) = O(|R_i| \log |S_i| - |R_i| \log n)$. For massive datasets, $n \ll |S_i|$, which implies this cost is $O(|R_i| \log |S_i|)$.

Hence, choice (1) is a natural pick for massive data. Our first challenge is how to create equal-sized blocks $\{D_1, \ldots, D_n\}$ such that $\forall x \in D_i$ and $\forall y \in D_j$ if $i < j$ then $x < y$, over a massive one-dimensional data set $D$ in the Map phase. Given a value $n$, if we know the $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$ quantiles of $D$, clearly, using these quantiles guarantees $n$ equal-sized blocks. However, obtaining these quantiles exactly is expensive for massive datasets in MapReduce.

Thus, we search for an approximation to estimate the $\phi$-quantile over $D$ for any $\phi \in (0,1)$ efficiently in MapReduce.

Let $D$ consist of $N$ distinct integers $\{d_1, d_2, \ldots, d_N\}$, such that $\forall i \in [1,N]$, $d_i \in \mathbb{Z}^+$, and $\forall i_1 \neq i_2$, $d_{i_1} \neq d_{i_2}$. For any element $x \in D$, we define the rank of $x$ in $D$ as $r(x) = \sum_{i=1}^{N}[d_i < x]$, where $[d_i < x] = 1$ if $d_i < x$ and 0 otherwise. We construct a sample $\widehat{D} = \{s_1, \ldots, s_{|\widehat{D}|}\}$ of $D$ by uniformly and randomly selecting records with probability $p = \frac{1}{\varepsilon^2 N}$, for any $\varepsilon \in (0,1)$; $\forall x \in \widehat{D}$, its rank $s(x)$ in $\widehat{D}$ is $s(x) = \sum_{i=1}^{|\widehat{D}|}[s_i < x]$ where $[s_i < x] = 1$ if $s_i < x$ and 0 otherwise.

**Theorem 2.** *$\forall x \in \widehat{D}$, $\widehat{r}(x) = \frac{1}{p}s(x)$ is an unbiased estimator of $r(x)$ with standard deviation $\leq \varepsilon N$, for any $\varepsilon \in (0,1)$.*

*Proof.* We define $N$ independent identically distributed random variables, $X_1, \ldots, X_N$, where $X_i = 1$ with probability $p$ and 0 otherwise. $\forall x \in \widehat{D}$, $s(x)$ is given by how many $d_i$'s such that $d_i < x$ have been sampled from $D$. There are $r(x)$ such $d_i$'s in $D$, each is sampled with a probability $p$. Suppose their indices are $\{\ell_1, \ldots, \ell_{r(x)}\}$. This implies:

$$s(x) = \sum_{i=1}^{|\widehat{D}|}[s_i < x] = \sum_{i=1}^{r(x)} X_{\ell_i}.$$

$X_i$ is a Bernoulli trial with $p = \frac{1}{\varepsilon^2 N}$ for $i \in [1,N]$. Thus, $s(x)$ is a Binomial distribution expressed as $B(r(x), p)$:

$$\mathbf{E}[\widehat{r}(x)] = \mathbf{E}[\frac{1}{p}s(x)] = \frac{1}{p}\mathbf{E}[s(x)] = r(x) \text{ and,}$$

$$\begin{aligned} \mathrm{Var}[\widehat{r}(x)] &= \mathrm{Var}(\frac{s(x)}{p}) = \frac{1}{p^2}\mathrm{Var}(s(x)) \\ &= \frac{1}{p^2}r(x)p(1-p) < \frac{N}{p} = (\varepsilon N)^2. \end{aligned}$$

$\square$

Now, given required rank $r$, we estimate the element with the rank $r$ in $D$ as follows. We return the sampled element $x \in \widehat{D}$ that has the closest estimated rank $\widehat{r}(x)$ to $r$, i.e:

$$x = \operatorname*{argmin}_{x \in \widehat{D}} |\widehat{r}(x) - r|. \tag{3.5}$$

**Lemma 2.** *Any $x$ returned by Equation* (3.5) *satisfies:*

$$\Pr[|r(x) - r| \leq \varepsilon N] \geq 1 - e^{-2/\varepsilon}.$$

*Proof.* We bound the probability of the event that $r(x)$ is indeed away from $r$ by more than $\varepsilon N$. When this happens, it means that no elements with ranks that are within $\varepsilon N$ to $r$ have been sampled. There are $2\lfloor \varepsilon N \rfloor$ such rank values. To simplify the presentation, we use $2\varepsilon N$ in our derivation. Since each rank corresponds to a unique element in $D$, hence:

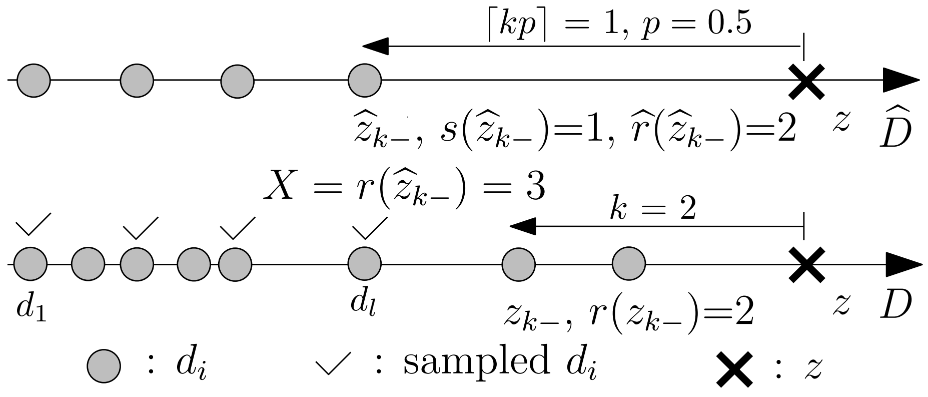
$$\begin{aligned} \Pr[|r(x) - r| > \varepsilon N] &= (1-p)^{2\varepsilon N} = (1 - \frac{1}{\varepsilon^2 N})^{2\varepsilon N} \\ &< e^{(-1/\varepsilon^2)2\varepsilon} = e^{-2/\varepsilon}. \end{aligned}$$

□

Thus, we can use $\widehat{D}$ and (3.5) to obtain a set of estimated quantiles for the $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$ quantiles of $D$. This means that for any $R_i$, we can partition $\mathrm{Z}_{R_i}$ into $n$ roughly equal-sized blocks for any $n$ with high probability.

However, there is another challenge. Consider the shifted copy $R_i$, and the estimated quantiles $\{z_{i,1}, \ldots, z_{i,n-1}\}$ from $\mathrm{Z}_{R_i}$. In order to partition $S_i$ using these $z$-values, we need to ensure that, in the $j$th block of $S_i$, $z^-(z_{i,j-1}, k, S_i)$ and $z^+(z_{i,j}, k, S_i)$ are copied over (as illustrated in Figure 3.4). This implies that the partitioning boundary of $S_{i,j}$ is no longer simply $z_{i,j-1}$ and $z_{i,j}$! Instead, they have been extended to the $k$th $z$-values to the left and right of $z_{i,j-1}$ and $z_{i,j}$ respectively. We denote these two boundary values for $S_{i,j}$ as $z_{i,j-1,k-}$ and $z_{i,j,k+}$. Clearly, $z_{i,j-1,k-} = \min(z^-(z_{i,j-1}, k, S_i))$ and $z_{i,j,k+} = \max(\ z^+(z_{i,j}, k, S_i))$; see Figure 3.4 for an illustration on the third $S_i$ block $S_{i,3}$. To find $z_{i,j-1,k-}$ and $z_{i,j,k+}$ exactly given $z_{i,j-1}$ and $z_{i,j}$ is expensive in MapReduce: we need to sort $z$-values in $S_i$ to obtain $Z_{S_i}$ and commute the entire $S_i$ to one reducer. This has to be done for every random shift.

We again settle for approximations using random sampling. Our problem is generalized to the following. Given a dataset $D$ of $N$ distinct integer values $\{d_1, \ldots, d_N\}$ and a value $z$, we want to find the $k$th closest value from $D$ that is smaller than $z$ (or larger than $z$, which is the same problem). Assume that $D$ is already sorted and we have the entire $D$, this problem is easy: we simply find the $k$th value in $D$ to the left of $z$ which is denoted as $z_{k-}$. When this is not the case, we obtain a random sample $\widehat{D}$ of $D$ by selecting each element in $D$ into $\widehat{D}$ using a sampling probability $p$. We sort $\widehat{D}$ and return the $\lceil kp \rceil$th value in $\widehat{D}$ to the left of $z$, denoted as $\widehat{z}_{k-}$, as our estimation for $z_{k-}$ (see Figure 3.5). When searching for $z_{k-}$ or $\widehat{z}_{k-}$, only values $\leq z$ matter. Hence, we keep only $d_i \leq z$ in $D$ and $\widehat{D}$ in the following analysis (as well as in Figure 3.5). For ease of discussion, we assume that $|D| \geq k$ and $|\widehat{D}| \geq \lceil kp \rceil$. The special case when this does not hold can be easily handled by returning the furthest element to the left of $z$ (i.e., $\min(D)$ or $\min(\widehat{D})$) and our theoretical analysis below for the general case can also be easily extended.



$\lceil kp \rceil = 1$, $p = 0.5$
$\widehat{z}_{k-}$, $s(\widehat{z}_{k-})$=1, $\widehat{r}(\widehat{z}_{k-})$=2
$z$
$\widehat{D}$
$X = r(\widehat{z}_{k-}) = 3$
$k = 2$
$d_1$
$d_l$
$z_{k-}$, $r(z_{k-})$=2
$z$
$D$
: $d_i$
: sampled $d_i$
: $z$

Figure 3.5: Estimate $z_{k-}$ given any value $z$.

Note that both $z_{k-}$ and $\widehat{z}_{k-}$ are elements from $D$. We define $r(d_i)$ as the rank of $d_i$ in *descending order* for any $d_i \in D$; and $s(d_i)$ as the rank of $d_i$ in *descending order* for any

$d_i \in \widehat{D}$. Clearly, by our construction, $r(z_{k-}) = k$ and $s(\widehat{z}_{k-}) = \lceil kp \rceil$.

**Theorem 3.** *Let $X = r(\widehat{z}_{k-})$ be a random variable; by our construction, $X$ takes the value in the range $[\lceil kp \rceil, r(d_1)]$. Then, for any $a \in [\lceil kp \rceil, r(d_1)]$,*

$$\Pr[X = a] = p \binom{a-1}{\lceil kp \rceil - 1} p^{\lceil kp \rceil - 1} (1-p)^{a - \lceil kp \rceil}, \tag{3.6}$$

$$\textit{and } \Pr[X - k] \leq \varepsilon N \geq 1 - e^{-2/\varepsilon} \textit{ if } p = 1/(\varepsilon^2 N). \tag{3.7}$$

*Proof.* When $X = a$, two events must have happened:

- $e_1$: the $a$th element $d_\ell$ $(r(d_\ell) = a)$ smaller than $z$ in $D$ must have been sampled into $\widehat{D}$.
- $e_2$: exactly $(\lceil kp \rceil - 1)$ elements were sampled into $\widehat{D}$ from the $(a-1)$ elements between $d_\ell$ and $z$ in $D$.

$e_1$ and $e_2$ are independent, and $\Pr(e_1) = p$ and $\Pr(e_2) = \binom{a-1}{\lceil kp \rceil - 1} p^{\lceil kp \rceil - 1}(1-p)^{a-1-(\lceil kp \rceil - 1)}$, which leads to (3.6).

Next, by Theorem 2, $\widehat{r}(x) = \frac{1}{p}s(x)$ is an unbiased estimator of $r(x)$ for any $x \in \widehat{D}$ (i.e., $\mathbf{E}(\widehat{r}(x)) = r(x)$). By our construction in Figure 3.5, $s(\widehat{z}_{k-}) = \lceil kp \rceil$. Hence, $\widehat{r}(\widehat{z}_{k-}) = k$. This means that $\widehat{z}_{k-}$ is always the element in $\widehat{D}$ that has the closest estimated rank value to the $k$th rank in $D$ (which corresponds exactly to $r(z_{k-})$). Since $X = r(\widehat{z}_{k-})$, when $p = 1/(\varepsilon^2 N)$, by Lemma 2, $\Pr[X - k] \leq \varepsilon N \geq 1 - e^{-2/\varepsilon}$. $\square$

Theorem 3 implies that boundary values $z_{i,j-1,k-}$ and $z_{i,j,k+}$ for any $j$th block $S_{i,j}$ of $S_i$ can be estimated accurately using a random sample of $S_i$ with the sampling probability $p = 1/(\varepsilon^2|S|)$: we apply the results in Theorem 3 using $z_{i,j-1}$ (or $z_{i,j}$ by searching towards the right) as $z$ and $S_i$ as $D$. Theorem 3 ensures that our estimated boundary values for $S_{i,j}$ will not copy too many (or too few) records than necessary ($k$ records immediately to the left and right of $z_{i,j-1}$ and $z_{i,j}$ respectively). Note that $z_{i,j-1}$ and $z_{i,j}$ are the two boundary values of $R_{i,j}$, which themselves are also estimations (the estimated $\frac{j-1}{n}$th and $\frac{j}{n}$th quantiles of $\mathrm{Z}_{R_i}$).

**The algorithm.** Complete *H-zkNNJ* is in Algorithm 2.

**Algorithm 2**: H-zkNNJ($R$, $S$, $k$, $n$, $\varepsilon$, $\alpha$)

```
1  get $\{\mathbf{v}_2, \dots, \mathbf{v}_\alpha\}$, $\mathbf{v}_i$ is a random vector in $\mathbb{R}^d$; $\mathbf{v}_1 = \overrightarrow{0}$;
2  let $R_i = R + \mathbf{v}_i$ and $S_i = S + \mathbf{v}_i$ for $i \in [1, \alpha]$;
3  for $i = 1, \dots, \alpha$ do
4      set $p = \frac{1}{\varepsilon^2|R|}$, $\widehat{R}_i = \emptyset$, $A = \emptyset$;
5      for each element $x \in R_i$ do
6          sample $x$ into $\widehat{R}_i$ with probability $p$;
7      $A$ =estimator1($\widehat{R}_i, \varepsilon, n, |R|$);
8      set $p = \frac{1}{\varepsilon^2|S|}$, $\widehat{S}_i = \emptyset$;
9      for each point $s \in S_i$ do
10         sample $s$ into $\widehat{S}_i$ with probability $p$;
11     $\widehat{z}_{i,0,k-} = 0$ and $\widehat{z}_{i,n,k+} = +\infty$;
12     for $j = 1, \dots, n-1$ do
13         $\widehat{z}_{i,j,k-}, \widehat{z}_{i,j,k+}$ =estimator2($\widehat{S}_i$, $k$, $p$, $A[j]$);
14     for each point $r \in R_i$ do
15         Find $j \in [1, n]$ such that $A[j-1] \le z_r < A[j]$;   Insert $r$ into the block $R_{i,j}$
16     for each point $s \in S_i$ do
17         for $j = 1 \dots n$ do
18             if $\widehat{z}_{i,j-1,k-} \le z_s \le \widehat{z}_{i,j,k+}$ then
19                 Insert $s$ into the block $S_{i,j}$
20 for any point $r \in R$ do
21     for $i = 1, \dots, \alpha$ do
22         find block $R_{i,j}$ containing $r$; find $C_i(r)$ in $S_{i,j}$;
23         for any $s \in C_i(r)$, update $s = s - \mathbf{v}_i$;
24     let $C(r) = \bigcup_{i=1}^{\alpha} C_i(r)$; output $(r, \text{knn}(r, C(r)))$;
```

**Algorithm 3**: estimator1($\widehat{R}$, $\varepsilon$, $n$, $N$)

```
1  $p = 1/(\varepsilon^2 N)$, $A = \emptyset$; sort $z$-values of $\widehat{R}$ to obtain $\mathrm{Z}_{\widehat{R}}$;
2  for each element $x \in \mathrm{Z}_{\widehat{R}}$ do
3      get $x$'s rank $s(x)$ in $\mathrm{Z}_{\widehat{R}}$;
4      estimate $x$'s rank $r(x)$ in $\mathrm{Z}_R$ using $\widehat{r}(x) = s(x)/p$;
5  for $i = 1, \dots, n-1$ do
6      $A[i]$=$x$ in $\mathrm{Z}_{\widehat{R}}$ with the closest $\widehat{r}(x)$ value to $i/n \cdot N$;
7  $A[0] = 0$ and $A[n] = +\infty$; return $A$;
```

```
Algorithm 4: estimator2($\widehat{S}$, $k$, $p$, $z$)
1 sort $z$-values of $\widehat{S}$ to obtain $Z_{\widehat{S}}$.
2 return the $\lceil kp \rceil$th value to the left and the $\lceil kp \rceil$th value to the right of $z$ in $Z_{\widehat{S}}$.
```

## 3.5 System Issues

We implement *H-zkNNJ* in three rounds of MapReduce.

**Phase 1:** The first MapReduce phase of H-zkNNJ corresponds to lines 1-13 of Algorithm 2. This phase constructs the shifted copies of $R$ and $S$, $R_i$ and $S_i$, also determines the partitioning values for $R_i$ and $S_i$. To begin, the master node first generates $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_\alpha\}$, saves them to a file in DFS, and adds the file to the distributed cache, which is communicated to all mappers during their initialization.

Each mapper processes a split for $R$ or $S$: a single record at a time. For each record $x$, a mapper, for each vector $\mathbf{v}_i$, first computes $\mathbf{v}_i + x$ and then computes the $z$-value $z_{\mathbf{v}_i+x}$ and writes an entry $(rid, z_{\mathbf{v}_i+x})$ to a single-chunk unreplicated file in DFS identifiable by the source dataset $R$ or $S$, the mapper's identifier, and the random vector identifier $i$. Hadoop optimizes a write to single-chunk unreplicated DFS files from a slave by writing the file to available local space. Therefore, typically the only communication in this operation is small metadata to the master. While transforming each input record, each mapper also samples a record from $R_i$ into $\widehat{R_i}$ as in lines 4-6 of Algorithm 2 and samples a record from $S_i$ into $\widehat{S_i}$ as in lines 8-10 of Algorithm 2. Each sampled record $x$ from $\widehat{R_i}$ or $\widehat{S_i}$ is emitted as a (($z_x$, $i$), ($z_x$, $i$, $src$)) key-value pair, where $z_x$ is a byte array for $x$'s $z$-value, $i$ is a byte representing the shift for $i \in [1, \alpha]$, and $src$ is a byte indicating the source dataset $R$ or $S$. We build a customized key comparator which sorts the emitted records for a partition (Hadoop's mapper output partition) in ascending order of $z_x$. We also have a customized partitioner which partitions each emitted record into one of $\alpha$ partitions by the shift identifier $i$ so all records from the $i$th shifted copy, both $\widehat{R_i}$ and $\widehat{S_i}$, end up in the same partition destined for the same reducer. Note each mapper only communicates sampled records to reducers.

In the reduce stage, a reduce task is started to handle each of the $\alpha$ partitions, consisting of records from $\widehat{R}_i$ and $\widehat{S}_i$. We define a grouping comparator which groups by $i$ to ensure each reducer task calls the reduce function only once, passing all records from $\widehat{R}_i$ and $\widehat{S}_i$ to the reduce function. The reduce function first iterates over each of the records from $\widehat{R}_i$ ($\widehat{S}_i$) in the ascending order of their $z$-values and saves these records into an array. The reducer estimates the rank of a record in $\mathrm{Z}_{R_i}$ if it is from $R_i$ as in lines 2-4 of Algorithm 3. Next, the reducer computes the estimated $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$ quantiles for $\mathrm{Z}_{R_i}$ from $\widehat{R}_i$, as in lines 5-6 of Algorithm 3. Note all records in $\widehat{R}_i$ are sorted by the estimated ranks so the reducer can make a single pass over $\widehat{R}_i$ to determine the estimated quantiles. After finding the $(n-1)$ estimated quantiles of $\mathrm{Z}_{R_i}$ the reducer writes these (plus $A[0]$ and $A[n]$ as in line 7 of Algorithm 3) to a file in DFS identifiable by $R_i$. They will be used to construct $R_{i,j}$ blocks in the second MapReduce phase. The final step of the reducer is to determine the partitioning $z$-values for $S_{i,j}$ blocks as in lines 11-13 of Algorithm 2 which are written to a file in DFS identifiable by $S_i$. The first phase is illustrated in Figure 3.6.

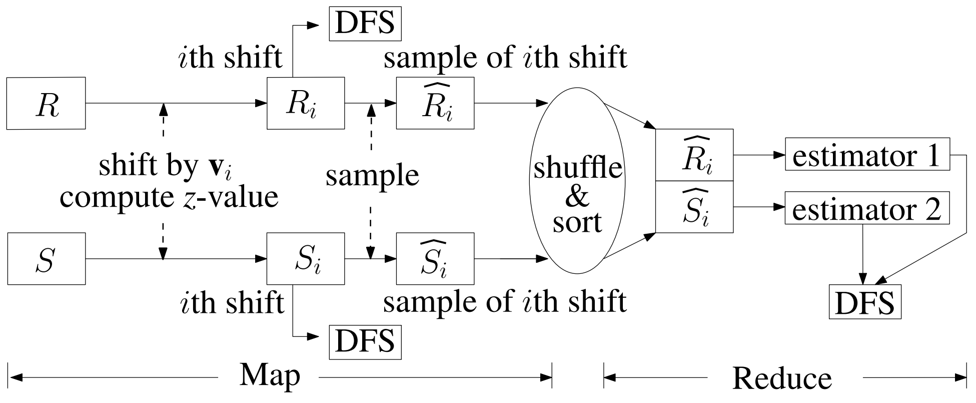


Figure 3.6: H-zkNNJ MapReduce Phase 1

**Phase 2**: The second phase corresponds to lines 14-19 for partitioning $R_i$ and $S_i$ into appropriate blocks, and then lines 20-23 for finding candidate points for $\text{knn}(r, S)$ for any $r \in R$, of Algorithm 2. This phase computes candidate set $C_i(r)$ for each record $r \in R_i$. The master first places the files containing partition values for $R_i$ and $S_i$ (outputs of reducers in Phase 1) into the distributed cache for mappers, as well as the vector file. Then, the master starts mappers for each split of the $R_i$ and $S_i$ files containing shifted records computed and written to DFS by mappers in phase 1.

Mappers in this phase emit records to one of the $\alpha n$ total $(R_{i,j}, S_{i,j})$ partitions. Mappers first read partition values for $R_i$ and $S_i$ from the distributed cache and store them in two arrays. Next, each mapper reads a record $x$ from $R_i$ or $S_i$ in the form $(rid, z_x)$ and determines which $(R_{i,j}, S_{i,j})$ partition $x$ belongs to by checking which $R_{i,j}$ or $S_{i,j}$ block contains $z_x$ (as in lines 14-19 of Algorithm 2). Each mapper then emits $x$, only once if $x \in R_i$ and possibly more than once if $x \in S_i$, as a key-value pair $((z_x, \ell), (z_x, rid, src, i))$ where $\ell$ is a byte in $[1, \alpha n]$ indicating the correct $(R_{i,j}, S_{i,j})$ partition. We implement a custom key comparator to sort by ascending $z_x$ and a custom partitioner to partition an emitted record into one of $\alpha n$ partitions based on $\ell$.

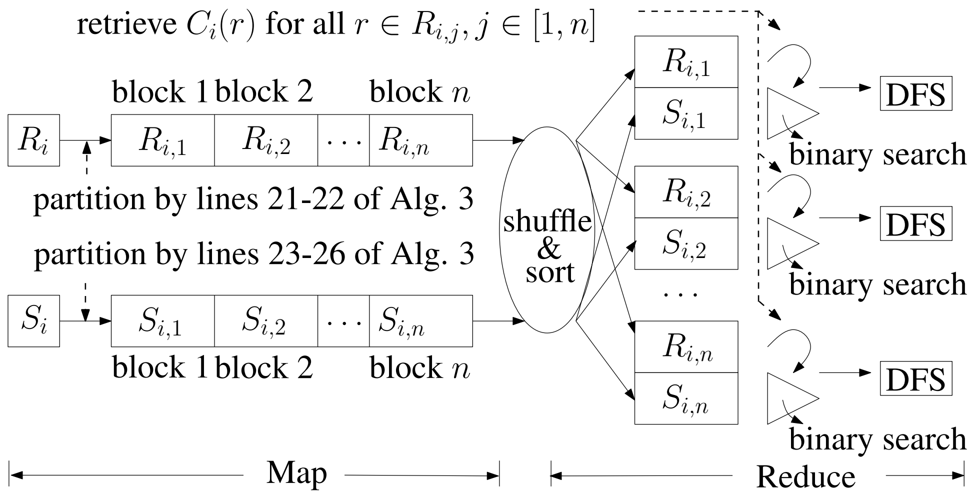

retrieve $C_i(r)$ for all $r \in R_{i,j}, j \in [1, n]$
block 1 block 2 block $n$
$R_i$
$R_{i,1}$ $R_{i,2}$ $\cdots$ $R_{i,n}$
partition by lines 21-22 of Alg. 3
partition by lines 23-26 of Alg. 3
$S_i$
$S_{i,1}$ $S_{i,2}$ $\cdots$ $S_{i,n}$
block 1 block 2 block $n$
shuffle & sort
$R_{i,1}$ $S_{i,1}$
$R_{i,2}$ $S_{i,2}$
$\cdots$
$R_{i,n}$ $S_{i,n}$
DFS
DFS
DFS
binary search
binary search
binary search
Map
Reduce

Figure 3.7: H-zkNNJ MapReduce Phase 2

The reducers compute $C_i(r)$ for each $r \in R_{i,j}$ in a partition $(R_{i,j}, S_{i,j})$; $\alpha n$ reducers are started, each handling one of the $\alpha n$ $(R_{i,j}, S_{i,j})$ partitions. A reducer first reads the vector file from the distributed cache. Each reduce task then calls the reduce function only once, passing in all records for its designated $(R_{i,j}, S_{i,j})$ partition. The reduce function then iterates over all records grouping them by their *src* attributes, storing the records into two vectors: one containing records for $R_{i,j}$ and the other containing records from $S_{i,j}$. Note in each vector entries are already sorted by their $z$-values, due to the sort and shuffle phase. The reducer next computes $C_i(r)$ for each $r \in R_{i,j}$ using binary search over $S_{i,j}$ (see Figure 3.1(b)), computes the coordinates from the $z$-values for any $s \in C_i(r)$ and $r$, then updates any $s \in C_i(r)$ as $s = s - \mathbf{v}_i$ and $r = r - \mathbf{v}_i$ and computes $d(r, s)$. The reducer then writes $(rid, sid, d(r, s))$ to a file in DFS for each $s \in \text{knn}(r, C_i(r))$. There is one such file per reducer. The second MapReduce phase is illustrated in Figure 3.7.

**Phase 3**: The last phase decides $\text{knn}(r, C(r))$ for any $r \in R$ from the $\text{knn}(r, C_i(r))$'s emitted by the reducers at the end of phase 2, which corresponds to line 24 of Algorithm 2. This can be easily done in MapReduce and we omit the details.

**Cost analysis.** In the first phase sampled records from $R_i$ and $S_i$ for $i \in [1, \alpha]$ are sent to the reducers. By our construction, the expected sample size is $\frac{1}{\varepsilon^2}$ for each $\widehat{R}_i$ or $\widehat{S}_i$, giving a total communication cost of $O(\frac{\alpha}{\varepsilon^2})$. For the second phase, we must communicate records from $R_i$ and $S_i$ to the correct $(R_{i,j}, S_{i,j})$ partitions. Each record $r \in R_i$ is communicated only once giving a communication cost of $O(\alpha|R|)$. A small percentage of records $s \in S_i$ may be communicated more than once since we construct a $S_{i,j}$ block using Lemma 1 and Theorem 3. In this case a $S_{i,j}$ block copies at most $2 \cdot (k + \varepsilon|S|)$ records from neighboring blocks with high probability giving us an $O(|S_{i,j}| + k + \varepsilon|S|)$ communication cost per $S_{i,j}$ block. In practice, for small $\varepsilon$ values we are copying roughly $O(k)$ values only for each block. For $\alpha n$ of the $S_{i,j}$ blocks, the communication is $O(\alpha \cdot (|S| + nk))$. Hence, the total communication cost is $O(\alpha \cdot (|S| + |R| + nk))$ for the second phase. In the final phase we communicate $\text{knn}(r, C_i(r))$ for each $r \in R_i$ giving us a total of $\alpha k$ candidates for each $r \in R$, which is $O(\alpha k|R|)$. Thus, the overall communication is $O(\alpha \cdot (\frac{1}{\varepsilon^2} + |S| + |R| + k|R| + nk))$. Since $\alpha$ is a small constant, $n$ and $k$ are small compared to $1/\varepsilon^2$, $|S|$ and $|R|$. The total communication of H-zkNNJ is $O(1/\varepsilon^2 + |S| + k|R|)$.

For the cpu cost, the first phase computes the shifted copies, then the partitioning values, which requires the samples $\widehat{R}_i$ and $\widehat{S}_i$ to be sorted (by $z$-values) and then they are processed by reducers. Further, for each identified partition value for $R_i$, we need to find its $\lceil kp \rceil$th neighbors (left and right) in $\mathrm{Z}_{\widehat{S}_i}$ to construct partition values for $S_i$, which translates to a cpu cost of $O(n \log(|\widehat{S}_i|))$. Over all $\alpha$ shifts, the total cost is $O(\alpha \cdot (|R| + |S| + |\widehat{R}| \log |\widehat{R}| + |\widehat{S}| \log |\widehat{S}| + n \log |\widehat{S}|))$, which is $O(\alpha \cdot (|R| + |S| + 1/\varepsilon^2 \log 1/\varepsilon^2 + n \log 1/\varepsilon^2))$. The cost of the second phase is dominated by sorting the $S_{i,j}$ blocks (in sort and shuffle) and binary-searching $S_{i,j}$ for each $r \in R_{i,j}$, giving a total cost of $O((|R_{i,j}| + |S_{i,j}|) \log |S_{i,j}|)$ for each $(R_{i,j}, S_{i,j})$ partition for every reducer. When perfect load-balancing is achieved, this translates to $O((\frac{|R|}{n} + \frac{|S|}{n}) \log \frac{|S|}{n})$ on each reducer. Summing costs over all $\alpha n$ of the $(R_{i,j}, S_{i,j})$ partitions we arrive at a total cost of $O(\alpha(|R| + |S|) \log \frac{|S|}{n})$ (or $O(\alpha(|R| + |S|) \log |S|)$ in the worst case if $S_i$ is not partitioned equally) for the second phase. The last phase can be done in $O(\alpha k |R| \log k)$. The total cpu cost for all three phases is $O(\alpha \cdot (\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon^2} + n \log \frac{1}{\varepsilon^2} + (|R| + |S|) \log |S| + k|R| \log k))$. Since $\alpha$ is a small constant and $k|R| \log k \ll (|R| + |S|) \log |S|$, the total cost is $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon^2} + n \log \frac{1}{\varepsilon^2} + (|R| + |S|) \log |S|)$.

**Remarks.** Our algorithm is designed to be efficient for small-medium $k$ values. Typical $k$NN join applications desire or use small-medium $k$ values. This is intuitive by the inherent purpose of ranking, i.e. an application is only concerned with the rank of the $k$ nearest neighbors and not about ranks of further away neighbors. When $k$ is large, in the extreme case $k = |S|$, the output size dictates no solution can do better than naive solutions with quadratic cost.

## 3.6 Experiments

### 3.6.1 Testbed and Datasets

We use a heterogeneous cluster consisting of 17 nodes with three configurations: (1) 9 machines with 1 Intel Xeon E5120 1.86GHz Dual-Core and 2GB RAM; (2) 6 machines with

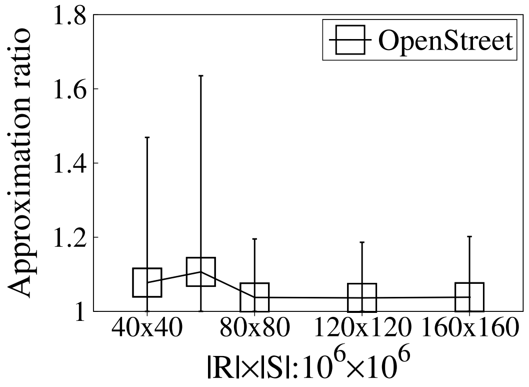
2 Intel Xeon E5405 2.00GHz Quad-Core processors and 4GB RAM; (3) 2 machines with 1 Intel Xeon E5506 2.13GHz Quad-Core processor and 6GB RAM. Each node is connected to a Gigabit Ethernet switch and runs Fedora 12 with hadoop-0.20.2. We select one machine of type (2) as the master node and the rest are used as slave nodes. The Hadoop cluster is configured to use up to 300GB of hard drive space on each slave and 1GB memory is allocated for each Hadoop daemon. One TaskTracker and DataNode daemon run on each slave. A single NameNode and JobTracker run on the master. The DFS chunk size is 128MB.

The default dimensionality is 2 and we use real datasets (OpenStreet) from the OpenStreetMap project [2]. Each dataset represents the road network for a US state. The entire dataset has the road networks for 50 states, containing more than 160 million records in 6.6GB. Each record contains a record ID, 2-dimensional coordinate, and description.

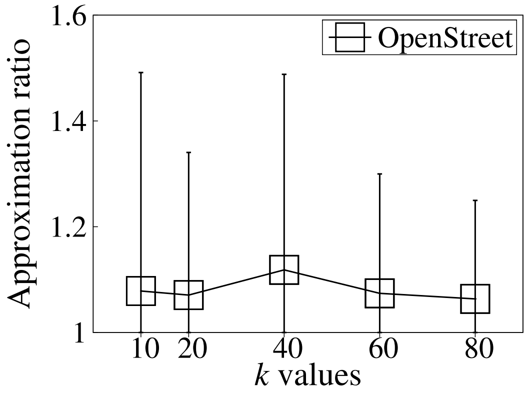
We also use synthetic Random-Cluster (R-Cluster) data sets to test all algorithms on datasets of varying dimensionality (up to 30). The R-Cluster datasets consist of records with a record ID and $d$-dimensional coordinates. We represent record IDs as 4-byte integers and the coordinates as 4-byte floating point types. We also assume any distance computation $d(r, s)$ returns a 4-byte floating point value.

A record's $z$-value is always stored and communicated as a byte array. To test our algorithms' performance, we analyze end-to-end running time, communication cost, and speedup and scalability. We generate a number of different datasets as $R$ and $S$ from the complete OpenStreet dataset (50 states) by randomly selecting 40, 60, 80, 120, and 160 million records. We use $(M \times N)$ to denote a dataset configuration, where $M$ and $N$ are the number of records (in ***million***) of $R$ and $S$ respectively, e.g., a $(40 \times 80)$ dataset has 40 million $R$ and 80 million $S$ records. Unless otherwise noted $(40 \times 40)$ OpenStreet is the default dataset.

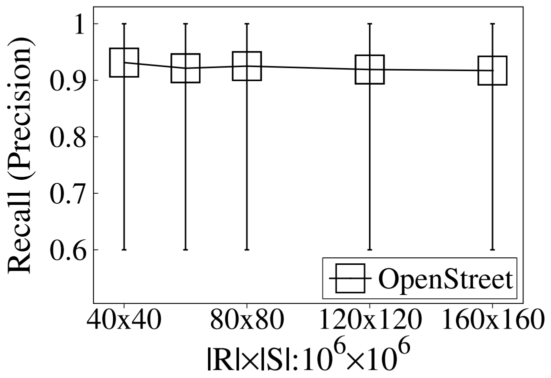
For H-zkNNJ, we use $\alpha = 2$ by default (only one random shift and original dataset are used), which already gives very good approximations as we show next. We use $\varepsilon = 0.003$ for the sampling methods by default. Let $\gamma$ be the number of physical slave machines used



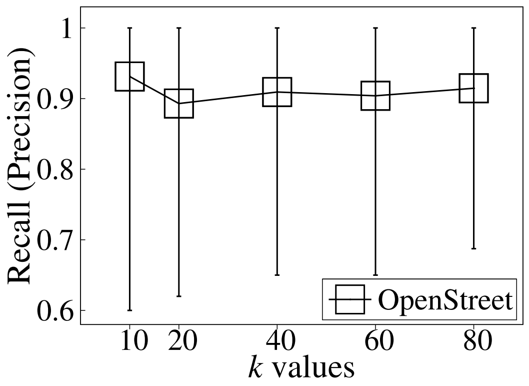
(a) Vary OpenStreet size.



(b) Vary $k$ on OpenStreet.



(c) Recall/Precision vs size.



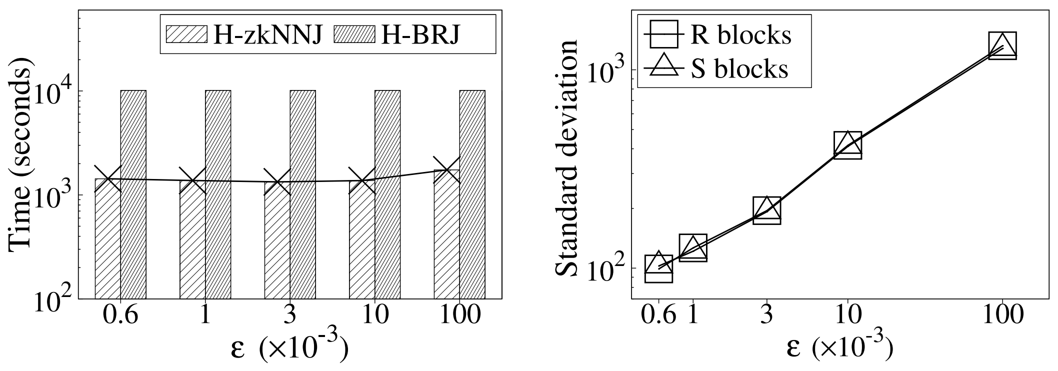
(d) Recall/Precision vs $k$.

Figure 3.8: Approximation quality of H-zkNNJ on (40x40) OpenStreet.

to run our methods in the cluster, we set the default number of reducers as $r = \gamma$. The number of blocks in $R$ and $S$ is $n = r/\alpha$ for H-zkNNJ and $n = \sqrt{r}$ for H-BNLJ and H-BRJ, since the number of buckets/partitions created by a mapper is $\alpha n$ in H-zkNNJ and $n^2$ in H-BNLJ and H-BRJ, which decides the number of reducers $r$ to run. In the default case, $\gamma = 16$, $k = 10$.

### 3.6.2 Performance Evaluation

**Approximation quality.** For our first experiment we analyze the approximation quality of H-zkNNJ. Since obtaining the exact $k$NN join results on the large datasets we have is very expensive, to study this, we randomly select 0.5% of records from $R$. For each of the selected records, we calculate its distance to the approximate $k$th-NN (returned by the H-zkNNJ) and its distance to the exact $k$th-NN (found by an exact method). The ratio between the two distances is one measurement of the approximation quality. We also measure the approximation quality by the *recall* and *precision* of the results returned by H-zkNNJ. Since both approximate and exact answers have exactly $k$ elements, its recall is equal to its precision in all cases. We plot the average as well as the 5% - 95% confidence interval for all randomly selected records. All quality-related experiments are conducted on a cluster with 16 slave nodes. Figures 3.8(a) to 3.8(d) present the results using the OpenStreet datasets. To test the influence of the size of datasets, we use the OpenStreet datasets and gradually increase datasets from (40x40) to (160x160) with $k = 10$. Figure 3.8(a) indicates H-zkNNJ exhibits excellent approximation ratio (with the average approximation ratio close to 1.1 in all cases and never exceeds 1.7 even in the worst case). This shows varying the size of datasets has almost no influence on the approximation quality of the algorithm. We next use (40x40) OpenStreet datasets to test how the approximation ratio is affected by $k$. Figure 3.8(b) indicates H-zkNNJ achieves an excellent approximation ratio with respect to $k$, close to 1.1 in all cases even when $k = 80$ and lower than 1.6 even in the worst case. Using the same setup, we also plot the recall and precision of H-zkNNJ when we vary the



(a) Running time. (b) Block size SD.

Figure 3.9: Effect of $\varepsilon$ in H-zkNNJ on (40x40).

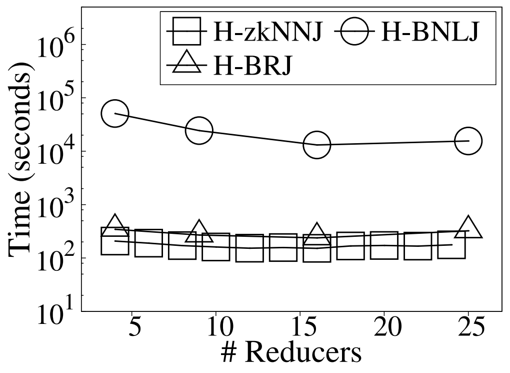
dataset sizes in Figure 3.8(c). Clearly, its average recall/precision is above 90% all the time; in the worst case, it never goes below 60%. Similar results on recall/precision hold when we vary $k$ as seen in Figure 3.8(d). We also extensively analyze the effect dimensionality $d$ and the number of shifts $\alpha$ have on the approximation quality in later experiments.

**Effect of $\varepsilon$.** Next we analyze the effect $\varepsilon$ has on the performance of H-zkNNJ using a (40x40) OpenStreet dataset. We notice in Figure 3.9(a) the running time of H-zkNNJ decreases as we vary $\varepsilon$ from 0.1 to 0.003, due to better load-balancing resulted from more equally distributed block sizes for $R_{i,j}$ and $S_{i,j}$. Decreasing $\varepsilon$ further actually causes the running time of H-zkNNJ to increase. This is because the additional time spent communicating sampled records (the sample size is $O(1/\varepsilon^2)$) in the first phase cancels out the benefit of having more balanced partitions in the second MapReduce phase. Regardless of the value of $\varepsilon$, H-zkNNJ is always an order of magnitude faster than H-BRJ. Figure 3.9(b) analyzes the standard deviation (SD) in the number of records contained in the $R_{i,j}$ and $S_{i,j}$ blocks in the reduce phase of the second MapReduce round. Note in the (40x40) dataset with

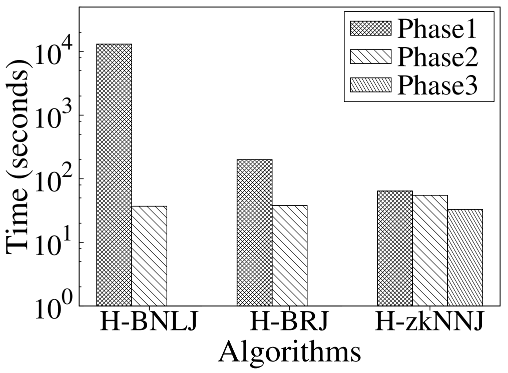
$n = \gamma/\alpha = 8$ in this case, ideally, each block (either $R_{i,j}$ or $S_{i,j}$) should have $5 \times 10^6$ records to achieve the optimal load balance. Figure 3.9(b) shows our sampling-based partition method is highly effective, achieving a SD from 100 to 5000 when $\varepsilon$ changes from 0.0006 to 0.1. There is a steady decrease in SD for blocks created from $R$ and $S$ as we decrease $\varepsilon$ (since sample size is increasing), which improves the load balancing during the second MapReduce phase of H-zkNNJ. However, this also increases the communication cost (larger sample size) and eventually negatively affects running time. Results indicate $\varepsilon = 0.003$ presents a good trade-off between balancing partitions and overall running time: the SD for both $R$ and $S$ blocks are about 200, which is very small compared to the ideal block size of 5 million. In this case, our sample is less than $120,000$ which is about 3‰ of the dataset size $4 \times 10^7$; $\varepsilon = 0.003$ is set as the default.

**Speedup and cluster size.** Next, we first use a (1x1) OpenStreet dataset and change the Hadoop cluster size to evaluate all algorithms. We choose a smaller dataset in this case, due to the slowness of H-BNLJ and to ensure it can complete in a reasonable amount of time. The running times and time breakdown of H-BNLJ, H-BRJ, and H-zkNNJ with varied cluster sizes $\gamma$ (the number of physical machines running as slaves in the Hadoop cluster) are shown in Figure 3.10. For H-zkNNJ, we test 7 cluster configurations, where the cluster consists of $\gamma \in [4, 6, 8, 10, 12, 14, 16]$ slaves and the number of reducers $r$ is equal to $\gamma$. We also test the case when $r > \gamma$ and $\gamma = 16$ (the maximum number of physical machines as slaves in our cluster) for $r \in [18, 20, 22, 24]$. We always set $n = r/\alpha$. For H-BNLJ and H-BRJ, we use 3 cluster configurations with $\gamma \in [4, 9, 16]$ and $r = \gamma$, hence $n = \sqrt{\gamma}$ (number of blocks to create in $R$ or $S$). We also test the case when $r > \gamma$ with $r = 25$, $\gamma = 16$, and $n = 5$.

We see in Figure 3.10(a) H-BNLJ is several orders of magnitude more expensive than either H-BRJ or H-zkNNJ even for a very small dataset. This is because we must compute $d(r, s)$ for a $r \in R$ against all $s \in S$ in order to find $\text{knn}(r, S)$, whereas in H-BRJ and H-zkNNJ we avoid performing many distance computations to accelerate the join processing.



(a) Running time.

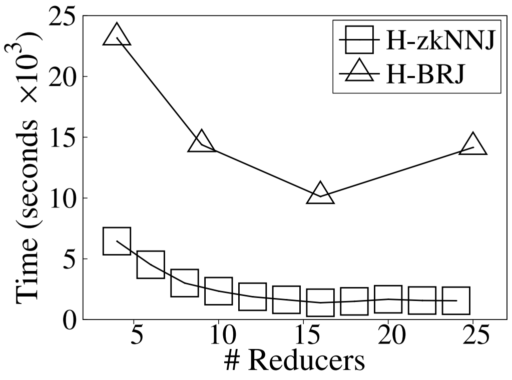


(b) Phase breakdown.

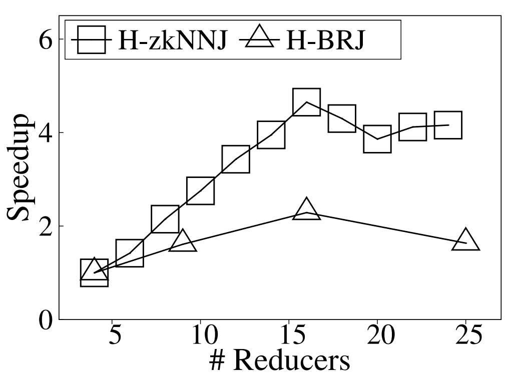
Figure 3.10: Running time for (1x1) OpenStreet.

The time breakdown of different phases of all algorithms with $r = \gamma = 16$ is depicted in Figure 3.10(b). A majority of the time for H-BNLJ is spent during its first phase performing distance calculations and communicating the $n^2$ partitions. We see utilizing the R-tree index in H-BRJ gives almost 2 orders of magnitude performance improvement over H-BNLJ in this phase. Clearly, H-BNLJ is only useful when H-BRJ and H-zkNNJ are not available and is only practical for small datasets (it gets worse when dataset size increases). Hence, we omit it from the remaining experiments.

Using the same parameters for $n$, $r$, $\gamma$ as above, we use a (40x40) OpenStreet dataset to further analyze the speedup and running times of H-zkNNJ and H-BRJ in varied cluster sizes. Figure 3.11(a) shows both running times reduce as the number of physical nodes increases. But as the number of reducers exceeds available slave nodes (which is 16) performance quickly deteriorates for H-BRJ whereas the performance decrease is only marginal for H-zkNNJ. This is because H-BRJ is dependent upon R-trees which require more time to construct and query. As the number of reducers $r$ for H-BRJ increases, the size of the $R$ and $S$ blocks decreases, reducing the number of records in each block of $S$. This means R-trees



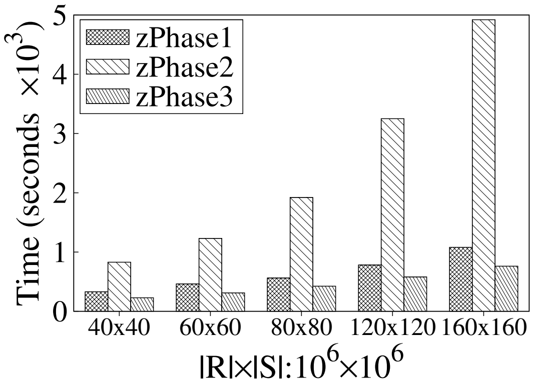
(a) Running time.



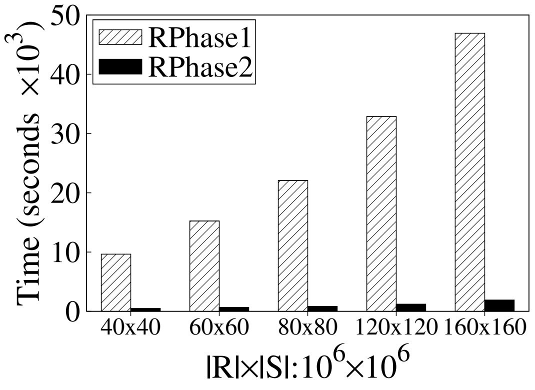
(b) Speedup.

Figure 3.11: Running time and speedup in (40x40).

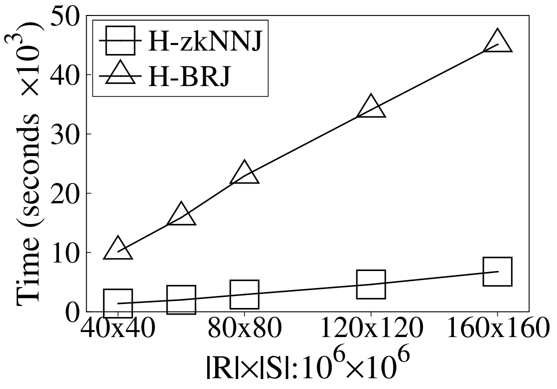
will be constructed over a smaller block. But we must construct more R-trees and we cannot construct these all in parallel, since there are more reducers than there are physical slaves. For this case, we see the reduced block size of $S$ is not significant enough to help offset the cost of constructing and querying more R-trees. The speedup of H-zkNNJ and H-BRJ are shown in Figure 3.11(b). The speedup for each algorithm is the ratio of the running time on a given cluster configuration over the running time on the smallest cluster configuration, $\gamma = 4$. In Figure 3.11(b), both H-zkNNJ and H-BRJ achieve almost a linear speedup up to $r = \gamma \leq 16$ (recall $r = \gamma$ for $\gamma \leq 16$, and $n = \sqrt{r}$ for H-BRJ and $n = r/\alpha$ for H-zkNNJ). Both algorithms achieve the best performance when $r = \gamma = 16$, and degrade when $r > 16$ (the maximum possible number of physical slaves). Hence, for remaining experiments we use $r = \gamma = 16$, $n = \sqrt{\gamma} = 4$ for H-BRJ, $n = \gamma/\alpha = 8$ for H-zkNNJ. Note H-zkNNJ has a much better speedup than H-BRJ when more physical slaves are becoming available in the cluster. From one reducer's point of view, the speedup factor is mainly decided by the block size of $R$ which is $|R|/n$, and $n = \sqrt{r}$ in H-BRJ, $n = r/\alpha = r/2$ in H-zkNNJ. Thus



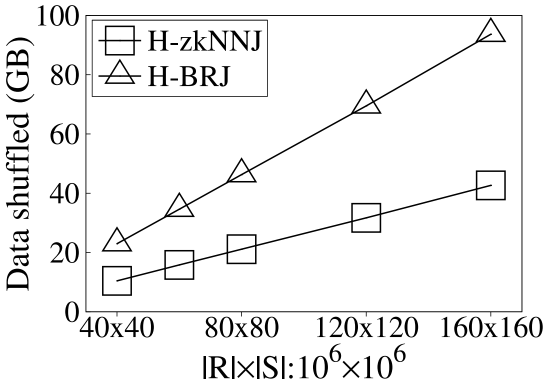
(a) H-zkNNJ breakdown.



(b) H-BRJ breakdown.

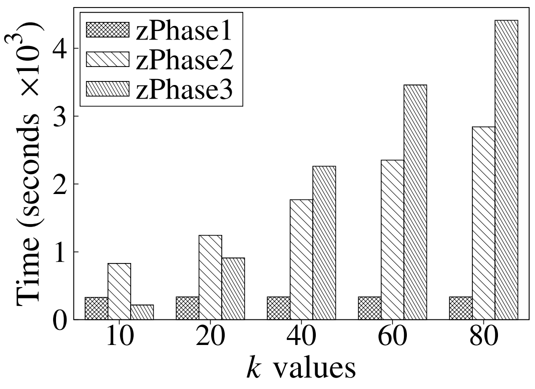


(c) Running time.

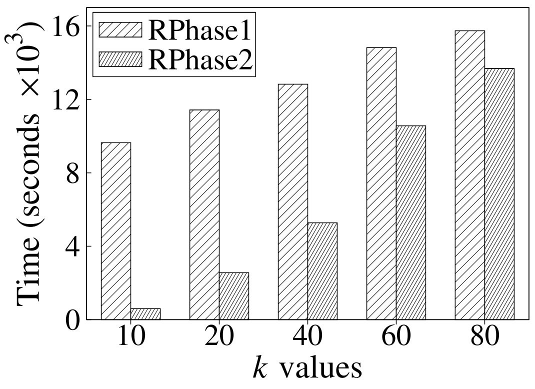


(d) Communication (GB).

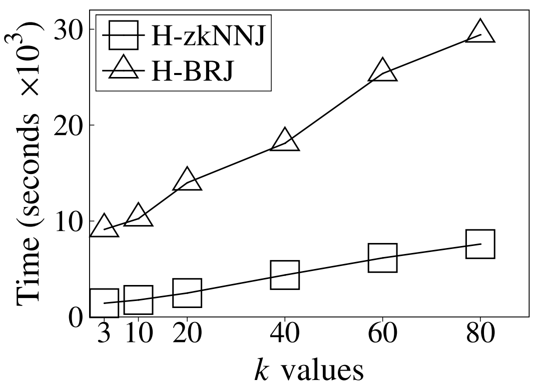
Figure 3.12: Phase breakdown, running time, and communication vs $|R| \times |S|$.



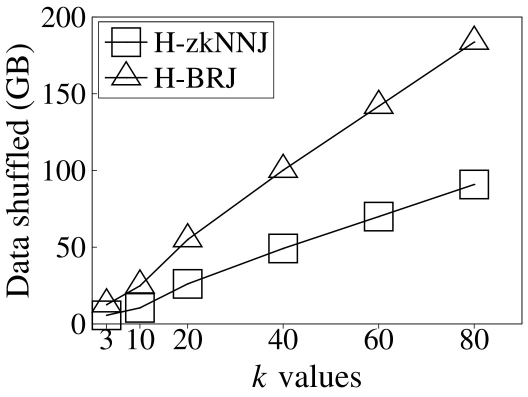
(a) H-zkNNJ breakdown.



(b) H-BRJ breakdown.



(c) Running time.



(d) Communication (GB).

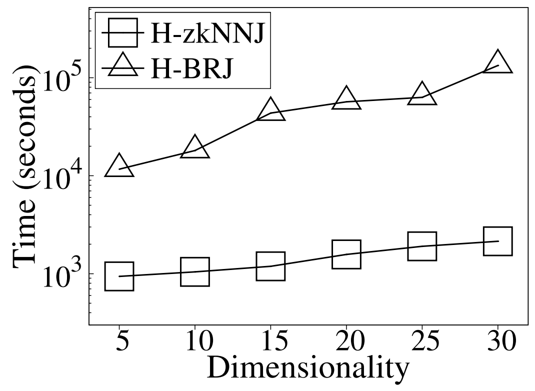
Figure 3.13: Phase breakdown, running time, and communication vs $k$ in OpenStreet.

the speedup of H-zkNNJ is $r/(2\sqrt{r})$ times more, which agrees with the trend in Figure 3.11(b) (e.g., when $r = \gamma = 16$, H-zkNNJ's speedup increases to roughly 2 times of H-BRJ's speedup). This shows not only H-zkNNJ performs better than H-BRJ, more importantly, the performance difference increases on larger clusters, by a factor of $O(\sqrt{\gamma})$.

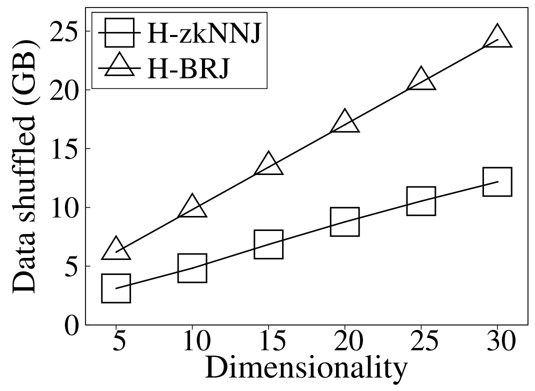
**Scalability.** Figures 3.12(a) and 3.12(b) demonstrate running times for different stages of H-zkNNJ and H-BRJ with different dataset configurations. We dub the three stages of H-zkNNJ zPhase1, zPhase2, and zPhase3 and the two stages of H-BRJ RPhase1 and RPhase2. The running time of each stage in both algorithms increases as datasets grow. Also, the second phase of H-zkNNJ and first phase of H-BRJ are most expensive, consistent with our cost analysis.

Clearly H-zkNNJ delivers much better running time performance than H-BRJ from Figure 3.12(c). The performance of H-zkNNJ is at least an order of magnitude better than H-BRJ when dealing with large datasets. The trends in Figure 3.12(c) also indicate H-zkNNJ becomes increasingly more efficient than H-BRJ as the dataset sizes increase. Three factors contribute to the performance advantage of H-zkNNJ over H-BRJ: (1) H-BRJ needs to duplicate dataset blocks to achieve parallel processing, i.e. if we construct $n$ blocks we must duplicate each block $n$ times for a total of $n^2$ partitions, while H-zkNNJ only has $\alpha n$ partitions; (2) Given the same number of blocks $n$ in $R$ and $S$, H-zkNNJ requires fewer machines to process all partitions for the $k$NN-join of $R$ and $S$ in parallel than H-BRJ does. H-zkNNJ needs $\alpha n$ machines while H-BRJ needs $n^2$ machines to achieve the same level of parallelism; (3) It takes much less time to binary-search one-dimensional $z$-values than querying R-trees. The first point is evident from the communication overhead in Figure 3.12(d), measuring the number of bytes shuffled during the Shuffle and Sort phases. In all cases, H-BRJ communicates at least 2 times more data than H-zkNNJ.

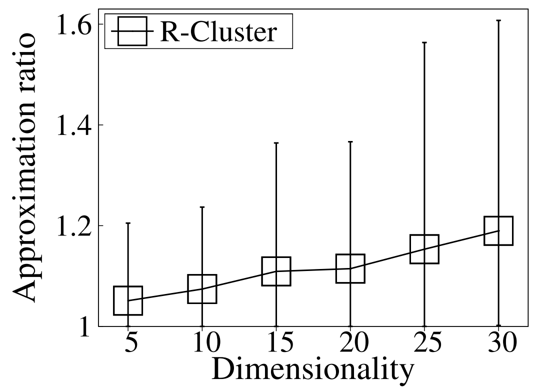
H-zkNNJ is highly efficient as seen in Figure 3.12(c). It takes less than 100 minutes completing $k$NN join on 160 million records joining another 160 million records, while H-BRJ takes more than 14 hours! Hence, to ensure H-BRJ can still finish in reasonable time,



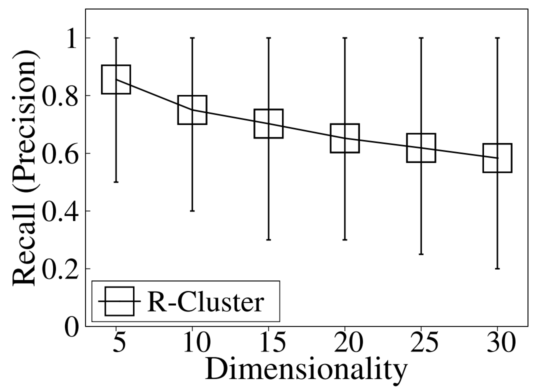
(a) Running time.



(b) Communication (GB).

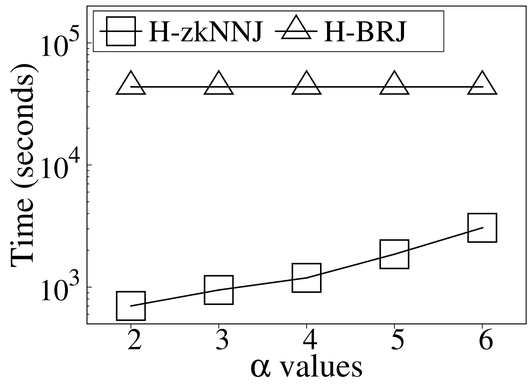


(c) Approximation ratio.

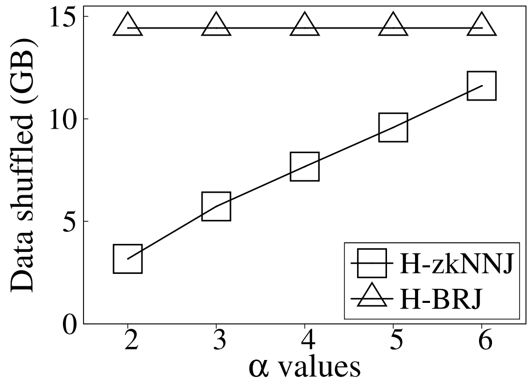


(d) Recall/Precision.

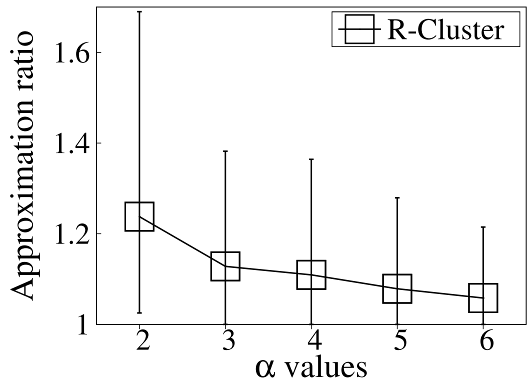
Figure 3.14: Running time, communication, and approximation quality vs $d$ in R-Cluster.



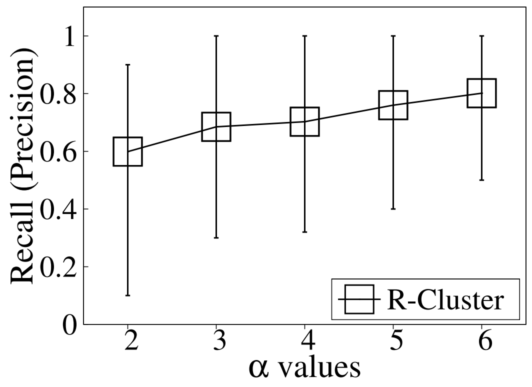
(a) Running time.



(b) Communication (GB).



(c) Approximation ratio.



(d) Recall/Precision.

Figure 3.15: Running time, communication, and approximation quality vs $\alpha$ in R-Cluster.

we use (40x40) by default.

**Effect of $k$.** Figures 3.13(a)–3.13(c) present running times for H-zkNNJ and H-BRJ with different $k$, using (40x40) OpenStreet datasets. Figure 3.13(a) shows zPhase1's running time does not change significantly, as it only performs dataset transformations and generates partition information, which is not affected by $k$. The execution time of zPhase2 and zPhase3 grow as $k$ increases. For larger $k$, H-zkNNJ needs more I/O and CPU operations in zPhase2 and incurs more communication overhead in zPhase3, which is similar to trends observed for RPhase1 and RPhase2 in Figure 3.13(b).

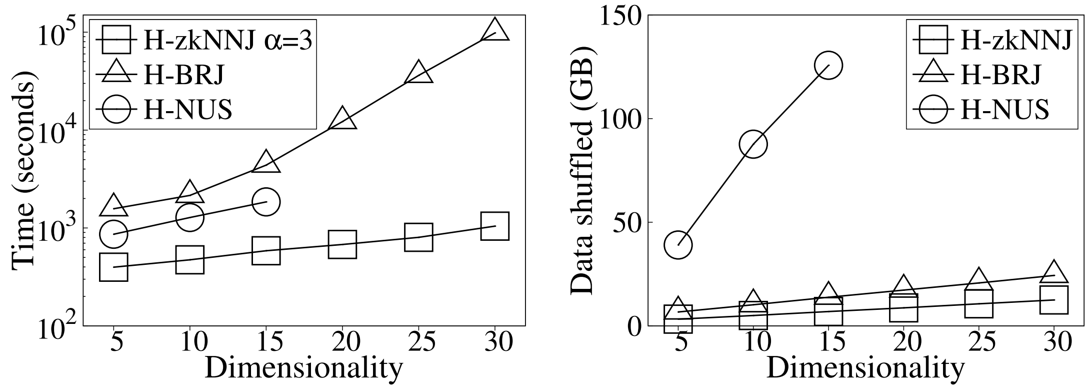
Figure 3.13(c) shows H-zkNNJ performs consistently (much) better than H-BRJ in all cases (from $k = 3$ to 80). For small $k$ values, $k$NN join operations are the determining factors for performance. For example, the performance of H-zkNNJ given by Figure 3.13(a) is mainly decided by zPhase2 where $k$ is 10. For large $k$, communication overheads gradually become a more significant performance factor for both H-zkNNJ and H-BRJ, evident especially when $k = 80$. We see in Figure 3.13(d) the communication for both algorithms increases linearly as $k$. However, H-BRJ requires almost 2 times as much communication as H-zkNNJ.

**Effect of dimension.** We generate (5x5) R-Cluster datasets with dimensionality $d \in [5, 10, 15, 20, 25, 30]$ (smaller datasets were used in high dimensions compared to the default two dimensional (40x40) OpenStreet dataset, to ensure that the exact algorithms can still finish in reasonable time). For these experiments, we also use one more random shift to ensure good approximation quality results in high dimensions. Figure 3.14 illustrates the running time, communication, and approximation quality of H-zkNNJ and H-BRJ on a cluster of 16 slave nodes with $k = 10$ and $\alpha = 3$ (2 random shifts plus the original dataset).

The experimental results in Figure 3.14(a) indicate H-zkNNJ has excellent scalability in high dimensions. In contrast the performance of H-BRJ degrades quickly with the increase of dimensionality. H-zkNNJ performs orders of magnitude better than H-BRJ, especially when $d \geq 10$. For example, when $d = 30$, H-zkNNJ requires only 30 minutes to compute the $k$NN join while H-BRJ needs more than 35 hours! In Figure 3.14(b) we see there is a linear relationship between communication cost and $d$. This is not surprising since higher dimensional data requires more bits to represent the $z$-values for H-zkNNJ and requires the communication of more 4-byte coordinate values for H-BRJ. In all cases H-BRJ communicates about 2 times more data than H-zkNNJ.

The results in Figures 3.14(c) and 3.14(d) demonstrate H-zkNNJ has good approximation quality for multi-dimensional data (up to $d = 30$) in terms of approximate ratio as well as recall and precision. The average approximation ratio in all cases are below 1.2 and even in the worst case when $d = 30$ the approximation ratio is only around 1.6 as indicated by Figure 3.14(c). As we can see from 3.14(d), the recall and precision drops slowly as $d$ increases, but the average recall and precision are still above 60% even when $d = 30$.

**Effect of random shift.** Finally, we investigate the effect the number of random shifts has on the performance and approximation quality of H-zkNNJ using (5x5) R-Cluster datasets on a cluster with 16 slave nodes. In the experiments, we vary the number of random



(a) Running time.

(b) Communication (GB).

Figure 3.16: Running time, communication vs $d$ in (5x5) R-Cluster, $\alpha = 3$.

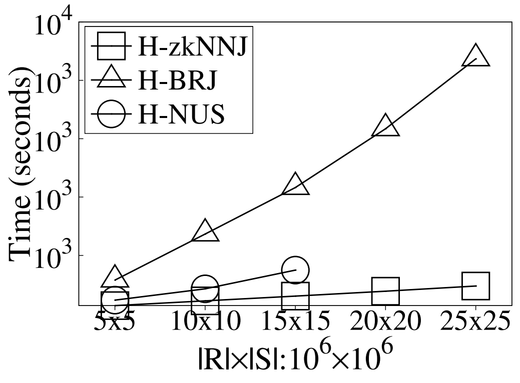
shifts $\alpha \in [2, 3, 4, 5, 6]$ and fix values of $d = 15$ and $k = 10$. Note that $\alpha$ shifts *imply* $(\alpha - 1)$ *random shifted copies plus the original dataset.*

Figures 3.15(a) and 3.15(b) indicate the performance time and communication cost of H-zkNNJ increase when we vary $\alpha$ from 2 to 6, due to the following reasons: (1) As $\alpha$ increases, $n$ (the number of partitions) of H-zkNNJ decreases as $n = r/\alpha$ and $r = \gamma$. As $n$ decreases the block size of $R$, which is $|R|/n$, being processed by each reducer increases. (2) As $\alpha$ grows, more random shifts have to be generated and communicated across the cluster, which leads to an increase in both performance time and communication cost. Nevertheless, H-zkNNJ always outperforms the exact method.

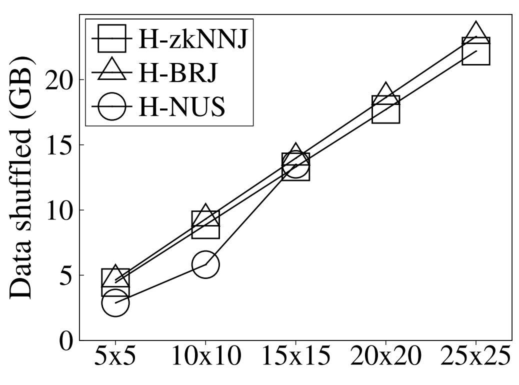
Figures 3.15(c) and 3.15(d) show that both the approximation ratio and recall/precision of H-zkNNJ improve as $\alpha$ increases. The average approximation ratio decreases from 1.25 to below 1.1 and the average recall/precision increases from 60% to above 80%.

**Performance comparisions with exact $k$NN join using MapReduce.** We compared the performance time of our proposed approximate algorithmsH-BRJ and H-zkNNJ [58] with that of the exact algorithm proposed in [34], dubbed as H-NUS, for multi-dimensional datasets using randomly generated datasets containing 5 million records with dimensionality varied from 5 to 30. Figure 3.16(a) indicates that H-NUS has better running time than the baseline H-BRJ solution, however, its running time is worse than that of H-zKNNJ. Specifically, H-NUS cannot efficiently process large size datasets and it runs out of memory when the dimensionality grows to 20. Figures 3.14(c) and 3.14(d) demonstrate that H-LSHZJ delivers excellent approximation quality both in approximation ratio and recall/precision.

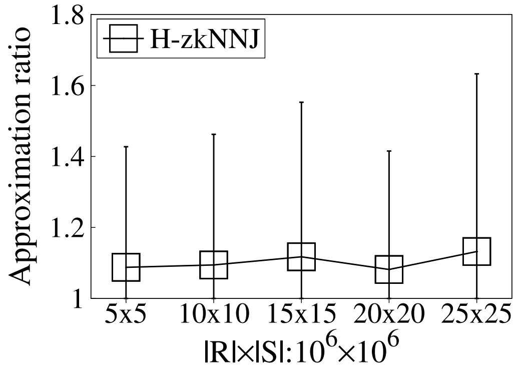
We also compare the performance among H-BRJ, H-zkNNJ, and H-NUS using OpenStreet datasets with configurations changing from (5x5) to (25x25) and the experimental results are given in Figure 3.17. Clearly, H-BRJ has the worst performance and H-zkNNJ ensures the best performance as in Figure 3.17(a). For datasets larger than (15x15), H-NUS cannot complete execution due to the lack of memory, indicating it cannot handle large datasets effectively. Figures 3.17(c) and 3.17(d) present the excellent approximation



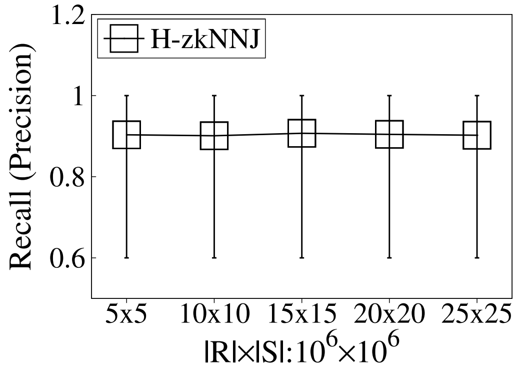
(a) Running time.



(b) Communication (GB).



(c) Approximation ratio.



(d) Recall/Precision.

Figure 3.17: Running time, communication, and communication vs $|R| \times |S|$.

ratio and recall/precision of H-zkNNJ.

**Remarks.** H-zkNNJ is the clear choice, when high quality approximation is acceptable, and requires only 2 parameters for tuning. The first parameter is the sampling rate $\varepsilon$ which determines how well balanced partitions are, hence the load balance of the system. Our experiments in Figure 3.9 show $\varepsilon = 0.003$ is already good enough to achieve a good load balance. Note $\varepsilon$ only affects load balancing and does not have any effect on the approximation quality of the join. The second parameter is the number of random shifts $\alpha$ which decides the approximation quality. By Theorem 1, using a small $\alpha$ returns a constant approximate solution on expectation and our experiments in Figures 3.8, 3.14(c), 3.14(d) and 3.15 verify this, showing $\alpha = 2$ already achieves good approximation quality in 2 dimensions, and $\alpha = 3$ is sufficient in high dimensions (even when $d$ increases to 30). Other parameters such as the number of mappers $m$, reducers $r$, and physical machines $\gamma$ are common system parameters for any MapReduce program which are determined based on resources in the cluster.

# CHAPTER 4

# PARALLEL $K$NN JOIN IN HIGH DIMENSIONS

## 4.1 Motivation

The $k$NN join algorithms for high-dimensional data or sparse data have been studied in [49, 53, 55] in traditional non-parallel environment. However, processing $k$NN related problems for high-dimensional data are mainly affected by the "curse of dimensionality" [7, 8], methods based on building multi-dimensional indexes cannot delivery good performance as the dimensionality of data sets goes beyond certain value and eventually the cost of $k$NN query is almost the same as naive sequential scan of the entire data set [8, 16]. In fact, [31] has verified that the existing popular multi-dimensional indexing methods such as Hybrid tree, $R^*$-tree, and iDistance are unable to efficiently deal with data in high-dimensional space. It has also been shown in [51] that all space partitioning based index techniques for $k$NN search degrade to linear search when dealing with high-dimensional data. On the other hand, researchers [5, 27, 30, 43] have pointed out that people can avoid dimensionality curse problem encountered in similarity search problem by using approximation instead of computing the exact solution. In fact, in many cases the approximate $k$NN result is almost as good as the exact result as long as the distance differences are small enough to be ignored. At the same time, there are few parallel $k$NN Join algorithms have been proposed for efficiently processing high-dimensional big data. Given all the above facts, we are motivated to design an approximate algorithm to efficiently solve $k$NN join problem for high-dimensional big data stored distributively using MapReduce framework.

## 4.2 Locality Sensitive Hashing

The concept of *locality sensitive hashing* (LSH) functions was first introduced by [19] to solve the NN ($k$NN) problem for high dimensional data. LSH functions can effectively preserving the locality information of points by ensuring any two points that near to each other have a higher collision probability than those that are far away from each other. Let $\mathcal{H}$ be a family of hash functions mapping high-dimensional space $R^d$ to one-dimensional space $U$. Given any two points $x, y \in R^d$, we can define the distance between $x$ and $y$ as $||x - y||$. Formally, we can have the following definition on LSH.

***locality sensitive hashing***. A function family $\mathcal{H}$ from $R^d$ to $U$ is called $(r, cr, p_1, p_2)$ sensitive if for any two points $x$,$y \in R^d$.

- if $||x, y|| \leq r$, then $Pr[h(x) = h(y)] \geq p_1$ ;
- if $||x, y|| > cr$, then $Pr[h(x) = h(y)] \leq p_2$.

Here, $r$ is a given distance, $c$ is the approximation ratio, and probabilities $p_1$, $p_2$ satisfy $p_1 > p_2$ such that LSH family $\mathcal{H}$ is useful in practice.

People have proposed various LSH families for different distance metrics. For $l_p$ norm, LSH families based on $p$-stable distributions are introduced by [19]. Specifically, a LSH function is defined as the following form:

$$h(o) = \lfloor \frac{a \cdot o + b}{w} \rfloor \tag{4.1}$$

,in which $o$ is a $d$-dimensional vector, $a$ is another $d$-dimensional vector with entries selected independently from a $p$-stable distribution, $b$ is a constant real number uniformly chosen from $[0, w)$, and $w$ is a large enough constant. In our work, we focus on the case $p = 2$, the Euclidean distance, and a normal distribution $N(0, 1)$ is a 2-stable distribution.

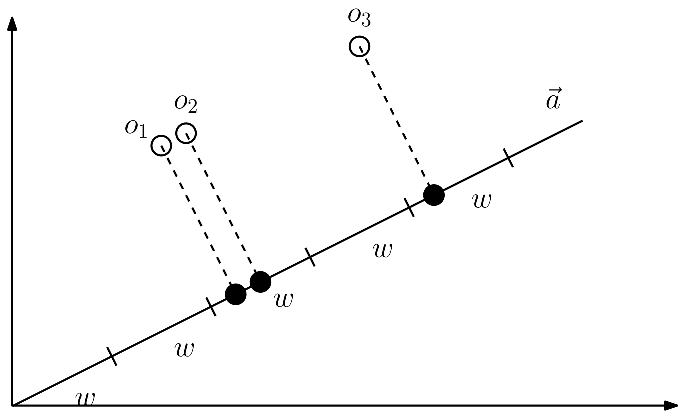


Figure 4.1: The interpretation of LSH.

The intuition of such a hash function is that, if two points are close to each other, their shifted projections (on line $\overrightarrow{a}$) will have the same hash value with high probability. On the other hand, two faraway points are likely to be projected into different values. In figure 4.1, points $o_1$, $o_2$ and $o_3$ are projected to the vector $\overrightarrow{a}$ (in this case, $b = 0$). The projections of $o_1$ and $o_2$ end up in the same interval while the projection of $o_3$ is in the different interval as it is far away from other two points.

LSH functions can be used for designing approximate NN($k$NN) algorithms, however, a single LSH function cannot guarantee a large enough difference between $p_1$ and $p_2$, which could lead to the low approximation quality results retrieved by similarity search algorithms. In practice, people amplify the difference by concatenating multiple LSH functions such that the collision probability of points far apart are really small. Particularly, we construct a family $\mathcal{G}$ of functions and each function $g \in \mathcal{G}$ has a form of $g(o) = (h_1(o), ..., h_m(o))$, where each $h_i$ is chosen independently and uniformly at random from $\mathcal{H}$. Then, $l$ functions $g_1, ..., g_l$ from $\mathcal{G}$ are chosen and used as the actual hash functions.

The main idea of basic LSH based $k$NN algorithms can generalized as the following two steps: Given a dataset $P$, we first create LSH-based indices using hash tables $\mathcal{T}_i, i \in [1, l]$, each of which is constructed by a function $g$ defined in $\mathcal{G}$. For a point $o$ in $P$, we compute its hash value $g_i(o)$ associated with the hash table $\mathcal{T}_i$ and add the point into the bucket $g_i(o)$ for every $i \in [1, l]$. After processing all points in $P$, we obtain a number of buckets, which can be used as indices for $k$NN search. Second, given a query point $q$, we locate all the buckets that $p$ will be hashed to and retrieve points from these buckets as candidate sets. Last, we calculate the distance between the query point $q$ and each point in the candidate sets and return top $k$ nearest points.

## 4.3 LSH based Solution

The *H-BNLJ* in [58] is straightforward and easy to implement exact $k$NN join solution, however, it suffers from excessive communication and computation overheads, therefore it cannot handle $k$NN join operation for large high-dimensional datasets. *H-zkNNJ* is effective solution for large multi-dimensional datasets, but it is also not very efficient for processing high-dimensional datasets.

This motivates us to find alternatives with linear communication and computation costs (to the number of records in each input dataset). To achieve this goal, we have to search for approximate solutions for high-dimensional data, leveraging on locality sensitive hashing (LSH). Using LSH to efficiently retrieve approximate nearest neighbors for high-dimensional data has been well studied in [22, 27, 36, 46]. These approaches manage to reduce the dimensionality of high-dimensional data to low-dimensional data while managing to preserve spatial locality, thus similarity search can be done efficiently on data with low dimension. Our proposed parallel algorithms are inspired by the above existing methods.

### 4.3.1 Overview of LSH KNN Join

The key advantage of LSH is to be able to preserve spatial locality for high-dimensional data set. The algorithm we proposed is based on the idea similar to [27], in which multiple LSH functions are used to ensure that points close to each other have a high probability of going to the same bucket. While points far away are unlikely to end up in the same bucket.

Our algorithm has two stages: candidate generation and candidate refinement stage. During the candidate generation stage, our algorithm divides $R$ and $S$ into a number of buckets using LSH-based functions such that each bucket contains points spatially close to each other. Consequently, for every point $p$ from $S$, its promising join candidates can be found in the buckets that $p$ falls into. In candidate refinement stage, for any given point $q$ from $R$, all $k$NN candidates of the point are examined by computing their distances to $q$, and the algorithm then returns the $k$ candidates with the least distance to $q$ as the point's $k$NN. In the following paragraphs, we first present more details on the candidate generation stage, then on candidate refinement stage.

The LSH-based hash functions used in the first stage are defined in $\mathcal{G}$, which is discussed in section 4.2. A function $g \in \mathcal{G}$ is a concatenation of $m$ LSH hash functions independently selected from $\mathcal{H}$. To ensure approximation quality of our algorithm, we randomly choose $l$ such functions $g_i$, $i \in [1..l]$ from $\mathcal{G}$. Here, $g_i$ is defined as $(h_{i,1}, ..., h_{i,m})$, where $h_{i,j} \in \mathcal{H}$ for $j$ $= 1$ to $m$. Basically, we create $l$ hash tables using $l$ hash functions $g_1, ..., g_l$ and group points from $R$ and $S$ into a number of buckets for each of $l$ hash tables. We generate $\{R_1, R_2, ..., R_l\}$ for $R$ and $\{S_1, S_2, ..., S_l\}$ for $S$ by applying $\{g_1, ..., g_l\}$ to $R$ and $S$. By computing hash value using $g_i$ for any record $r_0$ from $R$, then adding the value to the original record $r_0$ in $R$ with some associated information, we can construct $R_i$. ($S_i$ is constructed similarly.) Thus, we could separate records of $R_i$ and $S_i$ into buckets according their spatial similarities, i.e., records sharing the same $m$-dimensional hash value are to be placed into the same bucket. Given this fact, $\forall r \in R_i$, the candidate points from $S_i$, $C_i(r)$, for $r$'s $k$NN, can be found in the bucket containing $r$ using methods such as block nested loop join (BNLJ) without touching any other buckets. By now, we have a number of buckets which can be processed independently in parallel, however, we need to efficiently distribute these buckets over different computing nodes to ensure good performance. For a bucket $b$, we define its computational overhead $b.co$ as the multiplication of the number of points from $R_i$ and the number of points from $S_i$ in the bucket. We want to partition all buckets into $n$ partitions $\{P_1, ..., P_n\}$ and each partition $P_i$ has a workload $W_i$, which is defined as $\sum_{b \in P_i} b.co$, a sum of computational overheads of buckets in $P_i$. To achieve optimal performance, we want to minimize $W_i$, $i \in [1..n]$ and we adopt a greedy approximation algorithm, which sorts buckets by their corresponding computational overhead, then repeatedly assigns a bucket to a partition with the minimum workload until no buckets are left unassigned. During the candidate refinement stage, buckets in $\{P_1, ..., P_n\}$ can be processed in parallel. For $\forall r \in R_i$, it must appear once in a bucket belong to one of the $n$ total partitions, therefore, $C_i(r)$ can be conveniently calculated by comparing the distance between $r$ and all points from $S_i$ contained in the bucket and returning the top-$k$ closet points to $r$. In the end, we could obtain the knn$(r, S)$ by returning the top $k$ records from $C(r) = C_1(r) \cup ... \cup C_l(r)$.

We dub this approach as *H-LSHJ* (Hadoop LSH Join) and it is illustrated by Algorithm 5, which can be implemented by three MapReduce rounds in Hadoop.

### 4.3.2 System issues

In this section, we discuss implementation details of the three-round *H-LSHJ* in Hadoop as follows:

**Phase 1:** The first phase of H-LSHJ corresponds to lines 1-14 of Algorithm 5. This phase constructs $R_i$ and $S_i$, $i \in [1..l]$ from $R$ and $S$, then generates summary information for all possible buckets. The master node generates and writes parameters $a_i$ and $b_i$ to be used in $g_i \in \mathcal{G}$ into DFS files stored in the distributed cache of Hadoop, then it disseminates these parameters to all mappers during their initialization stages. It also initializes $B$, which contains buckets related information, as an empty set.

Every split of $R$ or $S$ invokes a map task, which processes a record by calling a map function. For reach record $r$ in the form of $(rid, c)$, the map function computes its $g_i(r)$, an $m$-dimensional vector and generates a record $(i, v, rid, c)$, which is to be written to a DFS file identifiable by $R_i$ or $S_i$ and $v = g_i(r)$. Hadoop optimizes a write to a DFS files from a slave by writing the file to available local space with high priority. Thus, only small meta-data to the master is usually involved in the communication in this operation. In practice, when datasets are large we could recompute $R_i$ or $S_i$ on-the-fly instead of writing them into DFS and reading them from DFS, which might be too expensive. At the same time, the mapper also generates an entry for the bucket, to which $r$ belongs, identified by the $v$ and $i$ by emitting a $((i), (i, v, src))$ key-value pair, where $i$ is a byte representing the hash table identifier for $i \in [1, l]$, $v$ is a byte array representing the bucket that $r$ goes into, and $src$ is a byte indicating the source dataset $R$ or $S$. We use a custom partitioner that partitions each emitted record into one of $l$ partitions by $i$ (the hash table identifier). As a result, all records from the same $i$th hash table, denoted as $T_i$, end up in the same partition. Lines 2-8 of Algorithm 5 are responsible for the above process.

In the Reduce stage, a reduce task is started to handle each of $l$ partitions, consisting of records from $T_i$, $i \in [1, l]$. We define a group comparator that groups records by their $v$ attributes to ensure each reduce task calls one reduce function for every different $v$. This means records belong to the same bucket $b$ identifiable by $v$ will be processed in a single reduce function. The reduce function maintains two counters $cr$ and $cs$, then it iterates over each record from $T_i$ belong to the same bucket $b$ and increases $cr$ or $cs$ by one depending on the record's $src$ attribute. At the end of reduce function, the reducer writes an entry $(i, v, cr, cs)$ representing the summary of the bucket $b$ to a DFS file, which is to be used in the second phone. After generating summary information of all buckets contained in $T_i$, the reduce task will terminate as in lines 9-14 of Algorithm 5.

**Phase 2:** In this stage, the algorithm first determines partitions of different buckets as in line 15 of Algorithm 5, it then dispatches records from $R_i$ or $S_i$ into their corresponding buckets, which are eventually routed to appropriate partitions via looking up the mapping between buckets and partitions as in lines 16-19 of Algorithm 5. The master node first decides the mapping between partitions and buckets using Algorithm 6 and saves the mapping as a DFS file, it then adds the mapping file into the distributed cache such that every mapper gets a copy of the mapping file during its starting stage.

Map tasks read partition information for buckets from the distributed cache and save these values into sets of $\{A_1,...,A_n\}$ corresponding to partitions $\{P_1,...,P_n\}$, respectively. Next, each map task invokes a map function for a record $x$ in the form ($i$, $v$, $rid$, $c$) from $R_i$ or $S_i$, the map function decides its destination bucket $b$ by $i$ and $v$ as well as its partition $\ell$ from $A_\ell$ containing $b$ as in lines 16-19 of Algorithm 5. Then, each mapper emits $x$ only once as a key-value pair $((i, v, \ell), (rid, src, c))$, where $\ell$ is a byte in $[1, n]$ indicating the right $P_j$ partition. We implement a custom partitioner to partition an emitted record into one of $n$ total partitions based on $\ell$.

For $r \in R$, its $C_i(r)$ could be found in the bucket $b$ identifiable by $i$ and $v = g_i(r)$. $b$ is contained in some $A_\ell$ and will be processed in partition $P_\ell$, $\ell \in [1, n]$. In the Reduce stage, $n$ reducers are started and each of them is responsible for exact one of $n$ $P_\ell$ partitions. We implement a custom group comparator grouping all records in the same partition by their $i$ and $v$ attributes. As a result, records are divided into a set of groups and the reduce task invokes a reduce function for each group from the group set. It is obvious that a group is equivalent to a bucket that we mentioned above. That is to say every bucket $b$ contained in $A_\ell$ will be handled by an independent reduce function. The reduce function iterates over all records grouping the by their $src$ attributes, storing the records into two arrays; one containing records from $R_i$, and the other containing records from $S_i$. Next, the reducer computes $C_i(r)$ for each $r \in R_i$ $(i \in [1, l])$ contained in the current group (bucket) using BNLJ method. Finally, the reducer writes $(rid, sid, d(r, s))$ to a file in DFS for each $s \in knn(r, C_i(r))$. This process is indicated by lines 20-23 of Algorithm 5.

**Phase 3:** The third phase decides $\text{knn}(r, C(r))$ for any $r \in R$ from the $\text{knn}(r, C_i(r))$'s emitted by the reducers at the end of phase 2, which corresponds to line 24 of Algorithm 5. This can be easily achieved by the following stages. In the Map stage, we read in all outputs from the second phase and use the unique record id of every record $r \in R$ as the partitioning key (at the end of the Map phase). Therefore, every reducer retrieves a $(rid, list(sid, d(r, s))$ pair and sorts the $list(sid, d(r, s))$ in the ascending order of $d(r, s)$. Then, the reducer emits the top-$k$ results for each $rid$.

**Algorithm 5**: H-LSHJ($R$, $S$, $k$, $l$, $m$, $w$)

```
generate $l$ functions $\{g_1, \dots, g_l\}$, where $g_i = (h_{i,1}, \dots, h_{i,m})$, $i \in [1, l]$ and $h_{i,j} \in \mathcal{H}$, $j \in [1, m]$;  $B = \emptyset$;
for $i = 1, \dots, l$ do
    for each point $r \in R$ do
        $v = g_i(r)$;  $src = 0$;  add $(i, v, r.rid, r.c)$ to $R_i$;
        add $(i, v, src)$ to $T_i$;
    for each point $s \in S$ do
        $v = g_i(r)$;  $src = 1$;  add $(i, v, s.sid, s.c)$ to $S_i$;
        add $(i, v, src)$ to $T_i$
    for each element $e$ in $T_i$ do
        locate or create $b \in B$ s.t. $b.i = e.i$ and $b.v = e.v$;
        if $e.src = 0$ then
            $b.cr = b.cr + 1$;
        else
            $b.sr = b.sr + 1$;
$\{A_1, \dots, A_n\}$ = partition($B$, $n$)
for $i = 1, \dots, l$ do
    for each record $x \in R_i$ or $S_i$ do
        locate $b \in A_\ell$, $1 \le \ell \le n$, s.t. $b.i = i$ and $b.v = x.v$;
        insert $x$ into $b$;
for any point $r \in R$ do
    for $i = 1, \dots, l$ do
        find the bucket $b$ in $A_\ell$, $\ell \in [1, n]$ containing $r$;
        find $C_i(r)$ in $b$;
    let $C(r) = \bigcup_{i=1}^{l} C_i(r)$;  output $(r, \mathrm{knn}(r, C(r)))$;
```

**Algorithm 6**: partition($B$, $n$)

```
for each element $b \in B$ do
    $b.co = b.r \times b.s$
sort elements in $B$ by their computational overheads;
for $i = 1, \dots, n$ do
    $W[i] = 0$;  $A_i = \emptyset$;
for $j = 1, \dots, |B|$ do
    $i$=Min($W$);  // return the index of the smallest value in W;
    $W[i] = W[i] + b_j.co$;
    add $b_j$ to $A_i$;
return $\{A_1, \dots, A_n\}$;
```

## 4.4 LSH and Z-value based KNN Join

One problem faced by the H-LSHJ algorithm is the load balancing problem. Considering the following examples given in Figure 4.2, Both $R$ and $S$ are consisted of clustered points, points in $R$ are represented as black points while points in $S$ are gray points. After we apply the LSH functions defined in equation 4.1 to these points, these points are projected onto vector $\vec{v}$. It is easy to observe that there is no single value of $w$ can fit for all points in $R$. For instance, to compute the 3NN for points located in $R$'s right cluster, $w_2$ is good choice, however, points in the left cluster of $R$ would not be able to find their 3NNs using $w_2$. One the other hand, if we choose $w_1$ to partition $R$ and $S$ into buckets, we end up with all points from $R$ falling into the same bucket, leading to serious unbalanced load.

To alleviate the above problem, we propose an efficient algorithm combining LSH and space-filling curves ($z$-order curve) together. Since both LSH and $z$-values have properties of effectively preserving the spatial locality of spatial points, we may approximate $k$NN join solutions by operating on one-dimensional values (transformed from high-dimensional points using LSH and $z$-order curve) instead of finding $k$NN join using high-dimensional points directly, leading to dramatic performance improvement. Specifically, the algorithm has the following steps: Firstly, we use LSH functions to reduce $d$-dimensional points to $m$-dimensional points ($d > m$). Secondly, we map $m$-dimensional points to one-dimensional values ($z$-values). Lastly, we are able to solve $k$NN joins by answering a number of single-dimensional range queries.

### 4.4.1 Algorithm Overview

The first step of this approach is to convert each point from $R$ and $S$ into a low-dimensional point using LSH-based functions. These LSH-based functions are from a function family $\mathcal{G}$ and a function $g \in \mathcal{G}$ can be expressed as $g(o) = (h_1(o), h_2(o), ..., h_m(o))$, where $h_j \in \mathcal{H}$, $j \in [1..m]$. As we discussed early, we can get an $m$-dimensional point $v = g(x)$ for a point $x$ in $R$ or $S$ using $g \in \mathcal{G}$ and $v$ can preserve the locality of $x$. Similar to *H-LSHJ* algorithm, to ensure high approximation quality of the proposed algorithm multi-

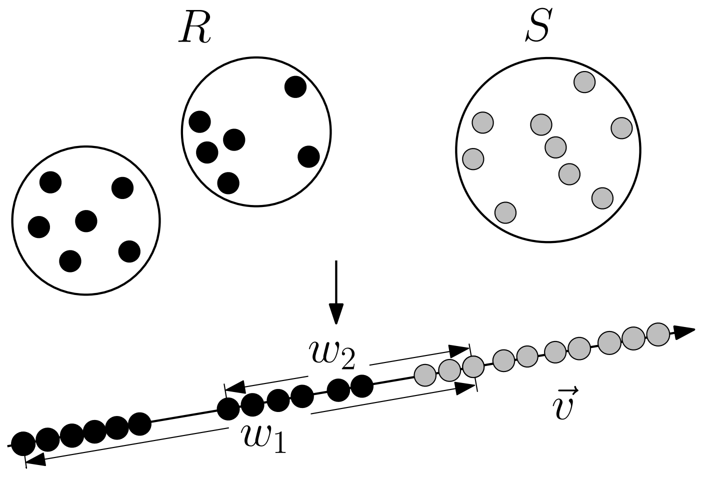


Figure 4.2: Clustered data set

ple LSH-based hash functions are necessary and we construct $l$ such hash functions $g_i \in \mathcal{G}$, $i \in [1, l]$. We could compute hash value $g_i(x)$ and add it to $x$ for every point $x$ in $R$ or $S$ for $i$ from 1 to $l$. As a result, $R$ and $S$ could be viewed as $m$-dimensional datasets after obtaining their low-dimensional hash values for every record in them. Thus, the problem becomes computing the $k$NN joins between low-dimensional datasets, which can be effectively solved using the similar approach as we have discussed in H-zkNNJ. Basically, for every $i \in [1, l]$, we could generate $\alpha$ random shift vectors and create $\alpha$ shifted copies of $R$ and $S$, leading to $\alpha l$ total copies $R_i$, $S_i$, $i \in [1, \alpha l]$ for $R$ and $S$. Next, we can compute the $z$-values of $R_i$ and $S_i$, partition $R_i$ and $S_i$ into partitions upon their $z$-values, and perform $k$NN join between $R_i$ and $S_i$ using techniques similar to those in H-zkNNJ. One modification to the H-zkNNJ algorithm is that we need to compute the actually distance between points using their $d$-dimensional coordinate instead of using their $m$-dimensional hash values. For $\forall r \in R_i$, let $C_i(r)$ be the $r$'s $k$NN candidate points from $S_i$ and the top-$k$ points from $C_1(r) \cup ... \cup C_l(r)$ can be used as $r$'s $k$NN.

The algorithm is given in Algorithm 7 and we dub this method as *H-LSHZJ*.

**Algorithm 7**: H-LSHZJ($R$, $S$, $k$, $n$, $\varepsilon$, $\alpha$, $l$, $m$, $w$)

```
1  get $l$ functions $\{g_1, \ldots, g_l\}$, where $g_{i'} = (h_{i',1}, \ldots, h_{i',m})$, $i' \in [1, l]$ and $h_{i',j'} \in \mathcal{H}$,
   $j' \in [1, m]$;
2  for $i' = 1, \ldots, l$ do
3      get $\{\mathbf{v}_{i',1}, \ldots, \mathbf{v}_{i',\alpha}\}$, $\mathbf{v}_{i',j}$ is a random vector in $\mathbb{R}^m$, $j \in [1..\alpha]$;  $R_i = S_i = \emptyset$;
4      for $j = 1, \ldots, \alpha$ do
5          let $i = (i' - 1) \times \alpha + j$;
6          for each record $r \in R$ do
7              let $v = g_{i'}(r) + \mathbf{v}_{i',j}$; add $(i, z_{\mathbf{v}_{i',j}}, r)$ to $R_i$;
8          for each record $s \in S$ do
9              let $v = g_{i'}(s) + \mathbf{v}_{i',j}$; add $(i, z_{\mathbf{v}_{i',j}}, s)$ to $S_i$;
10 for $i = 1, \ldots, \alpha l$ do
11     set $p = \frac{1}{\varepsilon^2 |S|}$, $\widehat{S}_i = \emptyset$;
12     for each element $x \in R_i$ do
13         sample $x$ into $\widehat{R}_i$ with probability $p$;
14     $A$ =estimator1$(\widehat{R}_i, \varepsilon, n, |R|)$;
15     set $p = \frac{1}{\varepsilon^2 |S|}$, $\widehat{S}_i = \emptyset$;
16     for each point $y \in S_i$ do
17         sample $y$ into $\widehat{S}_i$ with probability $p$;
18     $\widehat{z}_{i,0,k-} = 0$ and $\widehat{z}_{i,n,k+} = +\infty$;
19     for $j = 1, \ldots, n - 1$ do
20         $\widehat{z}_{i,j,k-}, \widehat{z}_{i,j,k+}$ =estimator2$(\widehat{S}_i,\ k,\ p,\ A[j])$;
21     for each point $r \in R_i$ do
22         Find $j \in [1, n]$ such that $A[j-1] \leq z_r < A[j]$;   Insert $r$ into the block $R_{i,j}$
23     for each point $s \in S_i$ do
24         for $j = 1 \ldots n$ do
25             if $\widehat{z}_{i,j-1,k-} \leq z_s \leq \widehat{z}_{i,j,k+}$ then
26                 Insert $s$ into the block $S_{i,j}$
27 for any point $r \in R$ do
28     for $i = 1, \ldots, \alpha l$ do
29         find block $R_{i,j}$ containing $r$;
30         find $C_i(r)$ in $S_{i,j}$;
31     let $C(r) = \bigcup_{i=1}^{\alpha} C_i(r)$; output $(r, \text{knn}(r, C(r)))$;
```

### 4.4.2 System issues

We implement *H-LSHZJ* in three rounds of MapReduce.

#### Phase 1:

The first MapReduce phase of H-LSHZJ corresponds to lines 1-20 of Algorithm 7. This

**Algorithm 8**: estimator1($\widehat{R}$, $\varepsilon$, $n$, $N$)

```
1 $p = 1/(\varepsilon^2 N)$, $A = \emptyset$; sort $z$-values of $\widehat{R}$ to obtain $Z_{\widehat{R}}$;
2 for each element $x \in Z_{\widehat{R}}$ do
3     get $x$'s rank $s(x)$ in $Z_{\widehat{R}}$;
4     estimate $x$'s rank $r(x)$ in $Z_R$ using $\widehat{r}(x) = s(x)/p$;
5 for $i = 1, \ldots, n-1$ do
6     $A[i]$=$x$ in $Z_{\widehat{R}}$ with the closest $\widehat{r}(x)$ value to $i/n \cdot N$;
7 $A[0] = 0$ and $A[n] = +\infty$; return $A$;
```

**Algorithm 9**: estimator2($\widehat{S}$, $k$, $p$, $z$)

```
1 sort $z$-values of $\widehat{S}$ to obtain $Z_{\widehat{S}}$.
2 return the $\lceil kp \rceil$th value to the left and the $\lceil kp \rceil$th value to the right of $z$ in $Z_{\widehat{S}}$.
```

phase reduces the dimensionality of the original data sets using LSH functions and constructs the copies of $R$ and $S$, $R_i$ and $S_i$ by building $z$-values upon their LSH hash values, also determines the partitioning values for $R_i$ and $S_i$. The master node first generates the necessary parameters such as $a_{i'}$ and $b_{i'}$ required by $g_{i'}$ and random vectors $\{\mathbf{v}_{i',1}, \mathbf{v}_{i',2}, ..., \mathbf{v}_{i',\alpha}\}$,$i' \in [1..l]$. These parameters and vectors are saved into DFS files, which are added to the distributed cache. Then, every mapper can get a copy of these DFS files upon its initialization stage .

Each mapper deals with a split of $R$ or $S$ and a map function is invoked for a single record at a time. For each record $r$ in $R$, the map function first computes its hash values $v = g_{i'}(r)$ (an $m$-dimensional vector) using LSH based function, then computes $\alpha$ random shift vector $v + \mathbf{v}_{i',j}$ and its corresponding $z$-value $z_{v+\mathbf{v}_{i',j}}$. Then, the mapper generates an entry ($i$, $z_{v+\mathbf{v}_{i',j}}$, $r$) and add it to $R_i$, which could be written to a single-chunk unreplicated file in DFS or be recomputed on-the-fly to save communication overhead depending on the situation. While transforming every input record, every mapper samples a record from $R_i$ into $\widehat{R}_i$ as in lines 11-13 of Algorithm 7 and also samples a record from $S_i$ into $\widehat{S}_i$ as in lines 15-17 of Algorithm 7. Every sampled record $x$ of $\widehat{R}_i$ or $\widehat{S}_i$ is emitted as a $((z_x, i), (z_x, i, src))$ key-value pair, where $z_x$ is a byte array for $x$'s $z$-value, $i$ is a byte denoting the shift for $i \in [1, \alpha l]$, and $src$ is a byte representing the source data set $R$ or $S$. We construct a custom key comparator sorting the emitted records for a partition (Hadoop's mapper output partition) in ascending order of $z_x$. A customized partitioner that partitions each emitted record into one of $\alpha l$ partitions by the shift identifier $i$. In this stage, every mapper only communicates sampled records to reducers.

In the Reduce stage, a reduce task is started to process each of the $\alpha l$ partitions, which is composed of records from $\widehat{R}_i$ and $\widehat{S}_i$. We build a custom grouping comparator that groups by $i$ to guarantee each reducer invokes the reduce function once and passes all records of $\widehat{R}_i$ or $\widehat{S}_i$ to the reduce function. The reduce function first iterates over each of the records from $\widehat{R}_i$ ($\widehat{S}_i$) in the ascending order of their $z$-values and saves these records into an array. The reducer estimates the rank of a record in $\mathrm{Z}_{R_i}$ if it is from $R_i$ as in lines 2-4 of Algorithm 8. Next, the reducer computes the estimated $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$ quantiles for $\mathrm{Z}_{R_i}$ from $\widehat{R}_i$, as in lines 5-6 of Algorithm 8. Note all records in $\widehat{R}_i$ are sorted by the estimated ranks so the reducer can make a single pass over $\widehat{R}_i$ to determine the estimated quantiles. After finding the $(n-1)$ estimated quantiles of $\mathrm{Z}_{R_i}$ the reducer writes these (plus $A[0]$ and $A[n]$ as in line 7 of Algorithm 8) to a file in DFS identifiable by $R_i$. They will be used to construct $R_{i,j}$ blocks in the second MapReduce phase. The final step of the reducer is to determine the partitioning $z$-values for $S_{i,j}$ blocks as in lines 18-20 of Algorithm 7 which are written to a file in DFS identifiable by $S_i$. The first phase is illustrated in Figure 4.3.

**Phase 2**: The second phase corresponds to lines 21-26 for partitioning $R_i$ and $S_i$ into appropriate blocks, and then lines 27-30 for finding candidate points for $\text{knn}(r, S)$ for any $r \in R$, of Algorithm 7. This phase computes candidate set $C_i(r)$ for each record $r \in R_i$. The master first places the files containing partition values for $R_i$ and $S_i$ (outputs of reducers in Phase 1) into the distributed cache for mappers, as well as the vector file. Then, the master starts mappers for each split of the $R_i$ and $S_i$ files containing shifted records computed and written to DFS by mappers in Phase 1.

Mappers in this phase would emit records to one of total $\alpha l n$ ($R_{i,j}$,$S_{i,j}$) partitions. Here, $R_{i,j}$ and $S_{i,j}$ have similar meanings as we discussed in H-zNNJ algorithm, which are

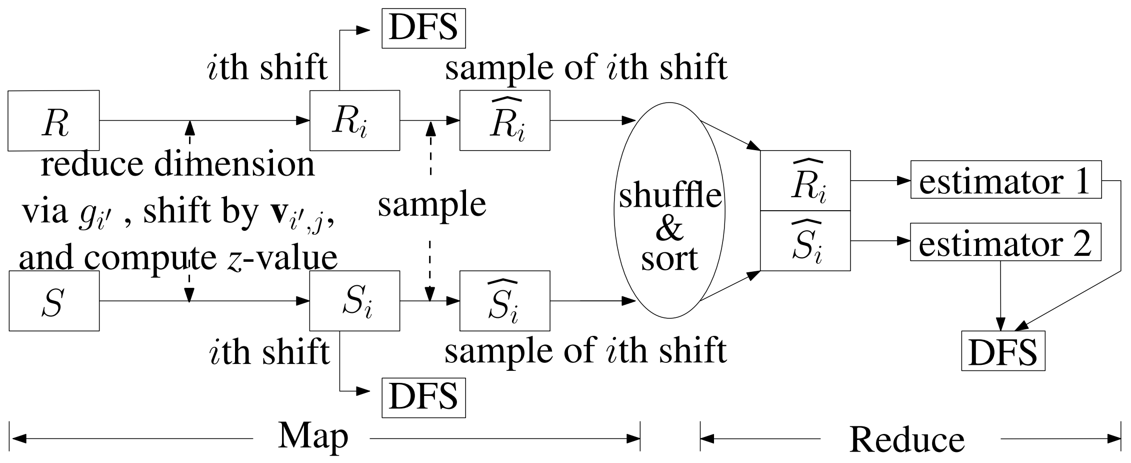


Figure 4.3: H-LSHZJ MapReduce Phase 1

blocks partitioned by $z$-values. Mappers first read partition values for $R_i$ and $S_i$ from the distributed cache and store them in two arrays. Next, each mapper reads a record $x$ from $R_i$ or $S_i$ in the form $(rid, z_x, c)$ and determines which $(R_{i,j}, S_{i,j})$ partition $x$ belongs to by checking which $R_{i,j}$ or $S_{i,j}$ block contains $z_x$ (as in lines 21-26 of Algorithm 7). Each mapper then emits $x$, only once if $x \in R_i$ and possibly more than once if $x \in S_i$, as a key-value pair $((z_x, \ell), (z_x, rid, src, i, c))$ where $\ell$ is a byte in $[1, \alpha ln]$ indicating the correct $(R_{i,j}, S_{i,j})$ partition. We implement a custom key comparator to sort by ascending $z_x$ and a custom partitioner to partition an emitted record into one of $\alpha ln$ partitions based on $\ell$.

The reducers compute $C_i(r)$ for each $r \in R_{i,j}$ in a partition $(R_{i,j}, S_{i,j})$; $\alpha ln$ reducers are started, each handling one of the $\alpha ln$ $(R_{i,j}, S_{i,j})$ partitions. Each reduce task then calls the reduce function only once, passing in all records for its designated $(R_{i,j}, S_{i,j})$ partition. The reduce function then iterates over all records grouping them by their $src$ attributes, storing the records into two vectors: one containing records for $R_{i,j}$ and the other containing records from $S_{i,j}$. Note in each vector entries are already sorted by their $z$-values, due to the sort and shuffle phase. The reducer next computes $C_i(r)$ for each $r \in R_{i,j}$ using binary search over $S_{i,j}$ Since each record contained in the $(R_{i,j}, S_{i,j})$ partition has its $d$-dimensional

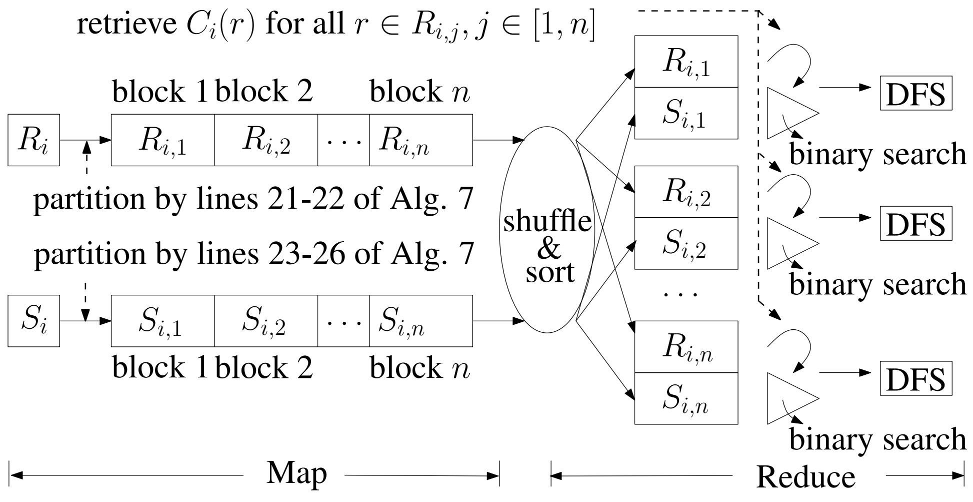


Figure 4.4: H-LSHZJ MapReduce Phase 2

coordinates denoted by its $c$ attribute. We can easily compute $d(r, s)$ for any $s \in C_i(r)$ and $r$. The reducer then writes $(rid, sid, d(r, s))$ to a file in DFS for each $s \in \text{knn}(r, C_i(r))$. There is one such file per reducer. The second MapReduce phase is illustrated in Figure 4.4.

**Phase 3**: The last phase decides $\text{knn}(r, C(r))$ for any $r \in R$ from the $\text{knn}(r, C_i(r))$'s emitted by the reducers at the end of phase 2, which corresponds to line 31 of Algorithm 7. This can be easily done in MapReduce and we omit the details.

## 4.5 Experiments

### 4.5.1 Testbed and Datasets

We use a cluster composed of 17 nodes with two configurations: (1) 1 SMP machine with 2 Intel Xeon E5645 Hexa-Core 2.40GHz , 128GB RAM, and 4TB HD; (2) 16 machines with 1 Intel Core i7-960 3.20GHz, 6GB RAM, and 2TB HD. All nodes are connected to a Gigabit Ethernet switch and run Ubuntu 10.04 with 2.6.32-45-server kernel. Each of 17 nodes are installed with java 1.6.0 and hadoop-1.0.3. We choose the machine with configuration (1) as the master node, which is used as both NameNode and JobTracker, and the total 16 nodes of configuration (2) as slave nodes, which also serve as DataNode. The major modifications to the Hadoop configurations are listed as follows. Each map and reduce task use up to 4GB RAM. Each node runs one map task and one reduce task. The speculative task execution option is disabled. The replication factor is set to 1. The default DFS chunk size is 256MB.

We extensively test the performance of our proposed algorithms using the following two datasets in our experiments:

**SIFT Data** This real dataset contains SIFT descriptors [33] with a dimensionality of 128 from the INRIA Holidays dataset [28]. The entire data INRIA Holidays set has more than 1 billion points and we randomly select up to 20 million points (around 13GB) from the whole dataset as our experimental datasets. We prepare the SIFT dataset such that each point in SIFT datasets consist of a record ID and $d$-dimensional coordinates. In our experiments, a record ID is treated as a 4-byte integer and the coordinates are of 4-byte floating point type.

**Cluster Data** This is a synthetic Cluster dataset to test algorithms on datasets with skewed distribution. Every record in the Cluster datasets have a record ID and 128-dimensional coordinate, which are represented in a similar way to SIFT dataset in our experiments.

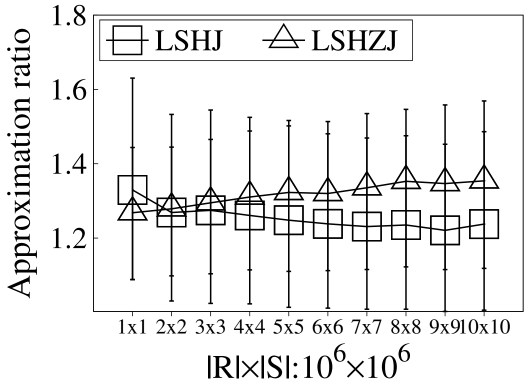
To measure the performance of our algorithms, we analyze the end-to-end running time, communication cost, and scalability. We use $(M \times N)$ to denote a dataset configuration, where $M$ and $N$ are numbers of records (in ***million***) of $R$ and $S$ respectively, e.g., a $(5 \times 5)$ dataset has 5 million $R$ and 5 million $S$ records. We construct 10 dataset configurations $(1 \times 1)$, ..., $(10 \times 10)$ for SIFT by randomly selecting various points from the total SIFT dataset and $(1 \times 1)$ Cluster dataset. $(1 \times 1)$ is default dataset configuration in most experiments.

For H-LSHZJ, we set the following default values: $\alpha = 2$, $k = 10$, $l = 16$, $m = 16$, and $k = 10$. For H-LSHJ, we set $l = 16$, $m = 5$, and $k = 10$ as default values. Let $\gamma$ be the number of physical slave machines used for running algorithms, we always set the default number of reducers $r = \gamma = 16$ for all algorithms.

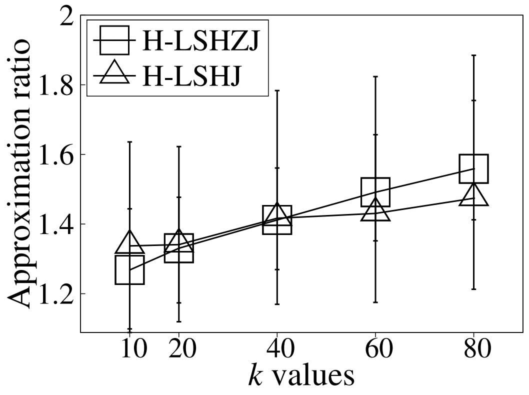
### 4.5.2 Performance Evaluation

**Approximation quality.** In this section, we study the approximation quality of both H-LSHJ and H-LSHZJ algorithms. Computing the exact $k$NN join results on large data with high dimensionality is extremely expensive, therefore, we randomly choose 0.5% of records from $R$. For every selected record, we calculate its distance to the approximate $k$th-NN obtained from an approximate approach (H-LSHJ or H-LSHZJ) and its distance to the exact $k$th-NN returned by an exact method. We define the ratio between the two distances as one of the metrics of the approximation quality. We also adopt the *recall* and *precision* of the results returned by approximate algorithms as additional quality metrics. In general cases, the approximate and exact answers have exactly $k$ elements, therefore, its recall is the same as its precision unless otherwise specified. We plot the average as well as the 5% - 95% confidence interval over all randomly selected points and the default cluster size is 16 slaves for all quality-related experiments. Figures 4.5 to 4.6 present the results using the SIFT datasets. To test the influence of the size of datasets over the approximation quality, we use the SIFT datasets and gradually increase datasets from (1x1) to (10x10) with $k = 10$. Figure 4.5(a) indicates both H-LSHJ and H-LSHZJ exhibits excellent approximation ratios (with the average approximation ratio around 1.33 in all cases and never exceeds 1.6 even in the worst case). This shows varying the size of datasets has very limited influence on the approximation quality for both algorithms.

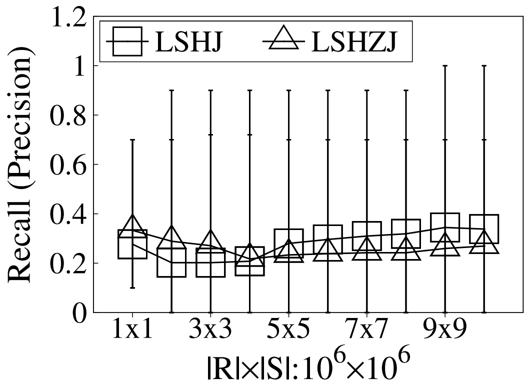
We next use (1x1) SIFT datasets to test how the approximation ratios of H-LSHZJ and H-LSHJ are affected by $k$, $l$, $m$, and $w$ as well as how different $\alpha$ might affect H-LSHZJ's approximation ratio. Figure 4.5(b) shows both H-LSHZJ and H-LSHJ algorithms achieve relative good approximation ratios with respect to $k$, only increases marginally even when $k$ grows to 80. As it is indicated by Figure 4.6(a), larger $\alpha$ could better preserve the locality of datasets, thus leads to lower approximation ratio. Figure 4.6(b) illustrates the approximation ratios of H-LSHZJ and H-LSHJ improve gradually as $l$ increases. Increasing $l$ is equivalent to use more LSH hash functions to process the datasets, resulting in better



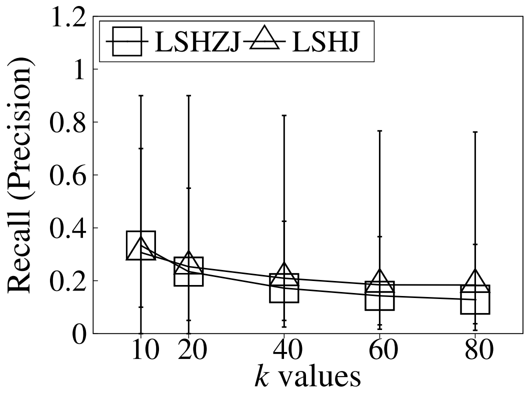
(a) Vary dataset size.



(b) Vary $k$.



(c) Recall/Precision vs size.



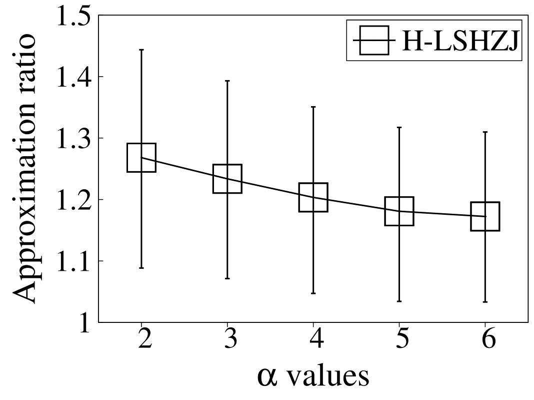
(d) Recall/Precision vs $k$.

Figure 4.5: Approximation quality of H-LSHJ and H-LSHZJ on SIFT (Part 1).

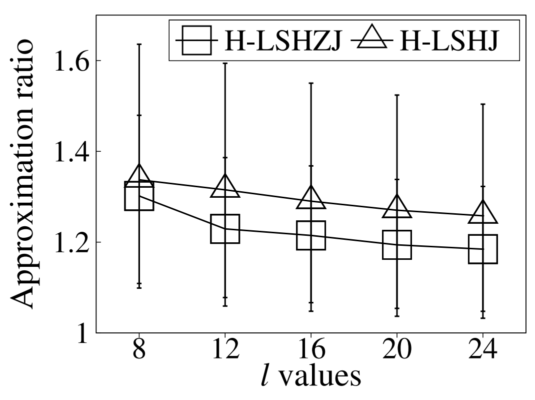
approximation quality. We use two different sets of $m$ values, (14, 15, 16, 17, 18) for H-LSHZJ and (3, 4, 5, 6, 7) for H-LSHJ to test how varying $m$ affects the approximation ratios of both algorithms. Figure 4.6(c) indicates the approximation ratio of H-LSHZJ becomes better with the increase of $m$, while the approximation ratio for H-LSHJ deteriorates rapidly as $m$ grows. For H-LSHJ algorithm, large $m$ values lead to many small size buckets such that $k$NN join result candidates in each bucket are limited, leading to poor approximate ratios. Varying $w$ has little influence over the approximation ratio of H-LSHZJ, however, it can greatly affect the approximation ratio of H-LSHJ. Large $w$ leads to buckets with large size, leading to huge approximation ratio improvement as in Figure 4.6(d) . In the extreme cases, all points could end up in one buckets. Using the same settings, we also plot the recall and precision of H-LSHJ and H-LSHZJ when we vary the size of datasets in Figure 4.5(c). As we can see that the average recalls/precisions for both algorithms are not good because it is high likely lots of points very close to a query points in high-dimensional space, resulting in poor recalls/precisions but high approximation ratios. However, in many cases, good approximate ratio is much more important than recall/precision and we could easily increase $l$ to greatly improve recall/precision of our proposed algorithms if necessary.

**Running time, communication cost, and scalability.** Figures 4.7(a) and 4.7(b) demonstrate the running times for different stages of H-LSHJ and H-LSHZJ with varied dataset configurations. The three stages of H-LSHZJ are dubbed as LZPhase1, LZPhase2, and LZPhase3 and the three stages of H-LSHJ are denoted by LPhase1, LPhase2, and LPhase3. It is obvious that the running time for each stage in both algorithms increases as datasets grow and the second phase of both H-LSHJ and H-LSHZJ is the most expensive stage because these two algorithms have to deal with the most computation and communication tasks in this stage.

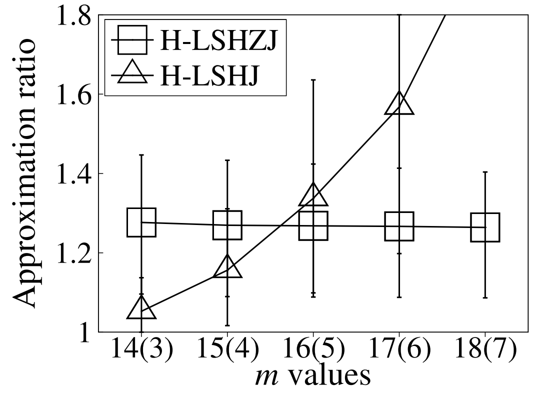
As we can see from Figure 4.7(c), H-LSHJ has slightly better running time than H-LSHZJ for datasets with less than 7 million points. However, H-LSHZJ delivers much better running time than H-LSHJ for datasets containing more than 8 million records indicating



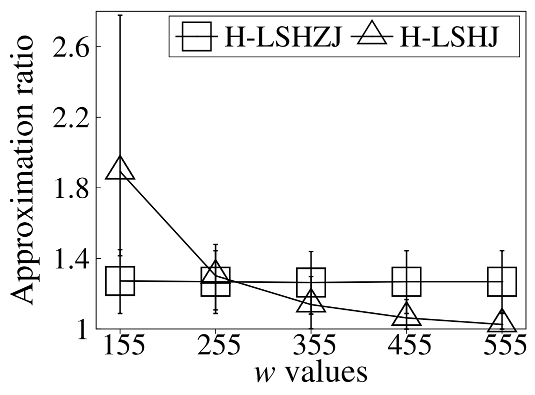
(a) Vary $\alpha$.



(b) Vary $l$.



(c) Vary $m$.



(d) Vary $w$.

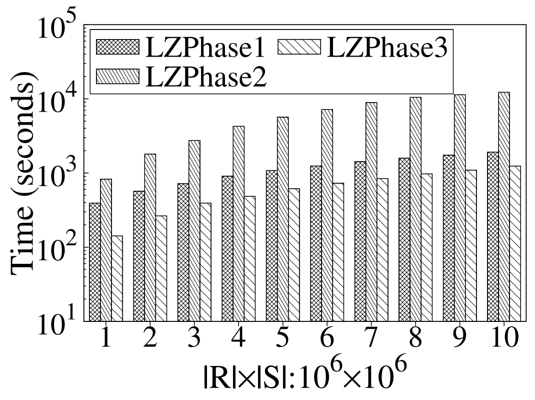
Figure 4.6: Approximation quality of H-LSHJ and H-LSHZJ on SIFT (Part 2).

that it is more efficient to deal with large datasets and has a better scalability than H-LSHJ. The reason is that H-LSHJ need to compute $k$NN join results for every bucket using BNLJ, whose running time grows rapidly as the number of buckets and the size of buckets increase for large size datasets. While for H-LSHZJ algorithm, it only needs to access several points around the query point (discussed in H-zkNNJ), therefore, its running time grows much slower than that of H-LSHJ.

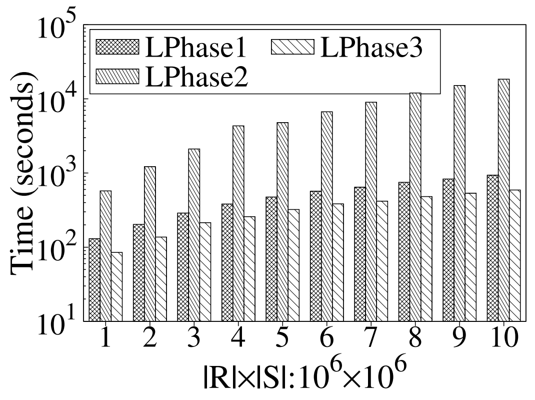
**Effect of $k$.** Figures 4.8(a)-4.8(c) present the running time for H-LSHJ and H-LSHZJ with different $k$ values, using (1x1) SIFT datasets. Figure 4.8(a) indicates LZPhase1's running time does not change significantly because it only transforms datasets and generates partition information, which is not affected by $k$. The similar trend can also be observed in Figure 4.8(b) showing LPHase1's running time is not affected by the values of $k$ as it only collects bucket-related information. The execution time of LZPhase2 and LZPhase3 as well as LPhase2 and LPhase3 increase as $k$ increases. For larger $k$, both H-LSHZJ and H-LSHJ need more I/O and CPU operations in LZPhase2 and LPhase2, leading to more communication overhead in LZPhase3 and LPhase3.

Figures 4.8(a) and 4.8(b) demonstrate that computing $k$NN join candidates is the deciding factors of the running time for both algorithms with small $k$ values (<=40). For example, the performance of H-LSHZJ illustrated by 4.8(a) is dominated by LZPhase2 when $k$ is 10. For large $k$, communication overheads gradually become another important performance factor for both H-LSHZJ and H-LSHJ. The communication for both algorithms increases linearly as $k$ as shown in 4.8(d).

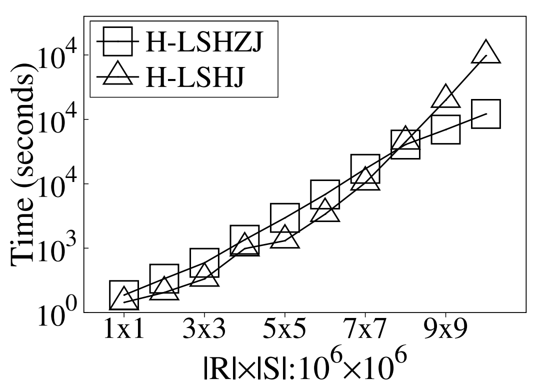
**Effect of $l$.** The running times of H-LSHZJ and H-LSHJ with varied $l$ values using (1x1) SIFT datasets are given by Figures 4.10(a) to 4.10(c). It is obvious that the running times of both algorithms increase linearly as $l$ grows. Figures 4.10(a) and 4.10(b) indicate that all stages of H-LSHJ and H-LSHZJ would take more time to complete with the increment of $l$. As $l$ determines the number of times that H-LSHZJ and H-LSHJ transform original high-dimensional datasets into low-dimensional datasets, therefore, large $l$ would incur



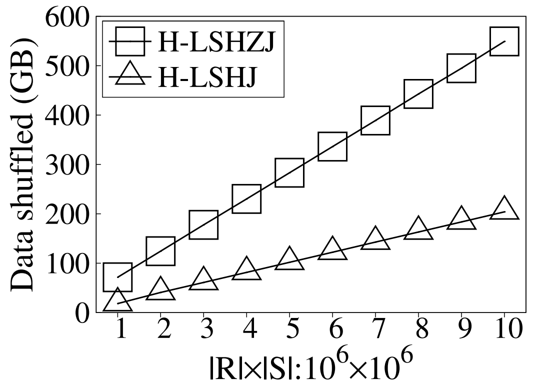
(a) H-LSHZJ breakdown.



(b) H-LSHJ breakdown.

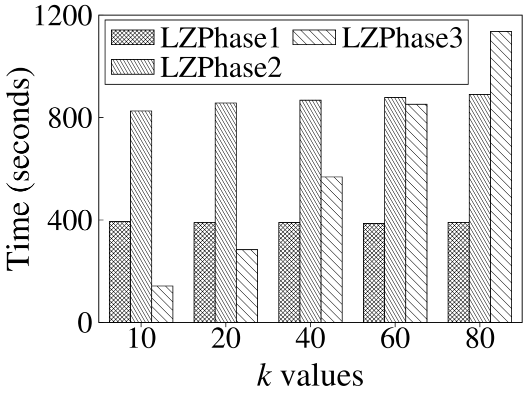


(c) Running time(1m).

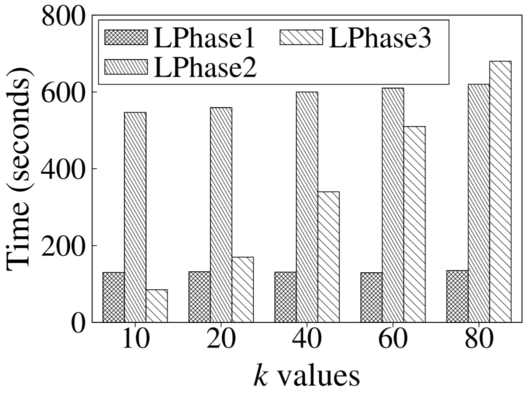


(d) Communication(1m).

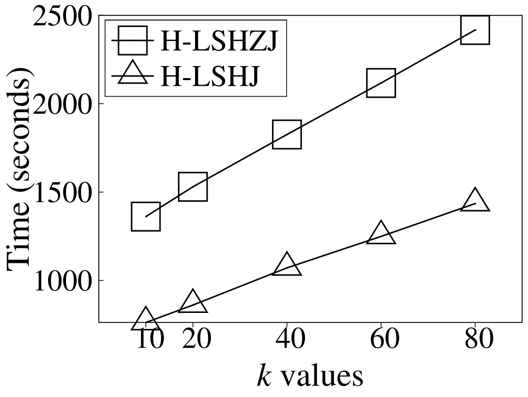
Figure 4.7: Phase breakdown, running time, and communication on SIFT.



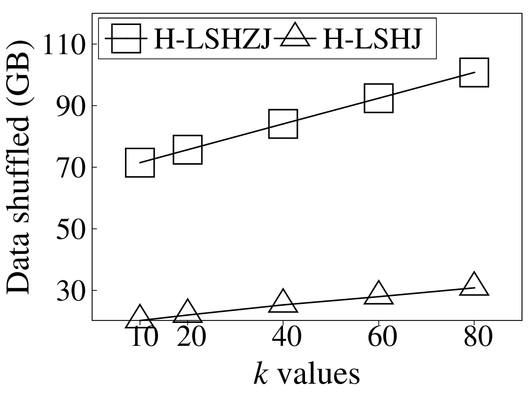
(a) H-LSHZJ breakdown.



(b) H-LSHJ breakdown.



(c) Running time.



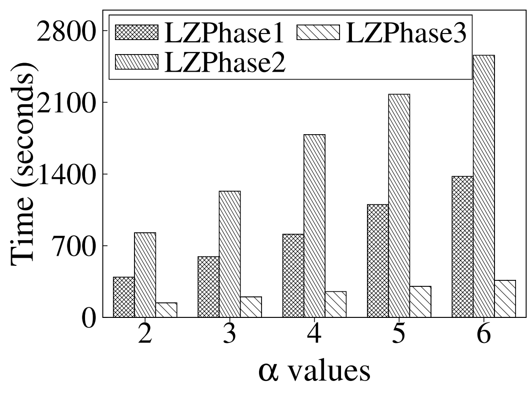
(d) Communication (GB).

Figure 4.8: Phase breakdown, running time, and communication vs $k$ in SIFT.

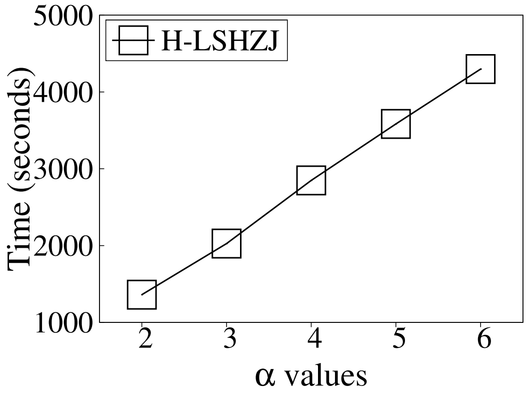
more computational and communication overhead in all stages of H-LSHZJ and H-LSHJ. With bigger $l$, take H-LSHJ as an example, it needs more time to collect bucket summary information associated with each of $l$ transformed datasets in LPhase1, to compute $l$ $k$NN join candidate sets, and to obtain the join result from the union of all candidate sets in LPhase2 and LPhase3, respectively. Figure 4.10(c) also shows that H-LSHJ takes less time than H-LSHZJ to finish because H-LSHZJ need to transform high-dimensional datasets into $l\alpha$ one-dimensional datasets while H-LSHJ only converts high-dimensional datasets into $l$ low-dimensional datasets. Figure 4.10(d) suggests that H-LSHJ and H-LSHZJ require more communication overhead as $l$ becomes larger. Also, H-LSHZJ incurs more communication overhead than H-LSHJ, which is consistent with the running time result.

**Effect of $m$.** Figures 4.11(a)-4.11(d) illustrate the running time and the communication consumption of H-LSHZJ and H-LSHJ with different $m$ values on (1x1) SIFT datasets. We vary $m$ from 14 to 18 for H-LSHZJ and $m$ from 3 to 7 for H-LSHJ. Clearly, the performance of H-LSHZJ is only slightly affected by $m$ and that of H-LSHJ can be greatly affected by $m$. For H-LSHJ, different $m$ values lead to variations the number of buckets in LPhase1 and LPhase2. The running time of H-LSHJ first decreases with the increase of $m$, then it grows rapidly with larger $m$. The reason is that smaller $m$ results in fewer buckets more points, and at the same time, larger $m$ values end up with more buckets with fewer points. The running time of H-LSHJ with small $m$ is high because the points in every bucket is large, requiring more computational time. On the other hand, the overhead of dealing with many small buckets gradually becomes the determining factor of the performance of H-LSHJ with increasing $m$, especially evident in Figure 4.11(c) when $m$ is 7. Different $m$ values have very limited influence on the communication overheads of both H-LSHJ and H-LSHZJ as shown in Figure 4.11(d).

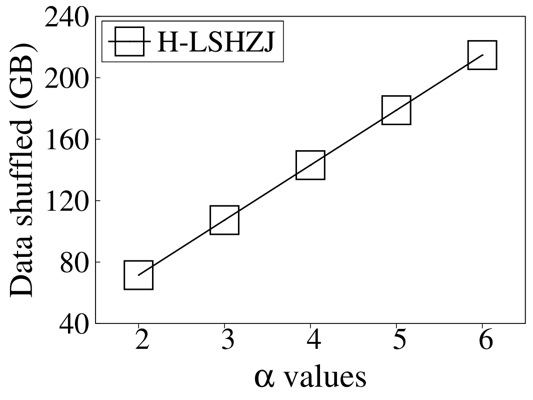
**Effect of $w$.** The influence of $w$ over the performance of H-LSHZJ and H-LSHJ are shown in Figures 4.12(a)-4.12(d). We can observe that the running times of the two approaches with varying $w$ have a similar trend as varying $m$ discussed in previous section. The number



(a) H-LSHZJ breakdown.



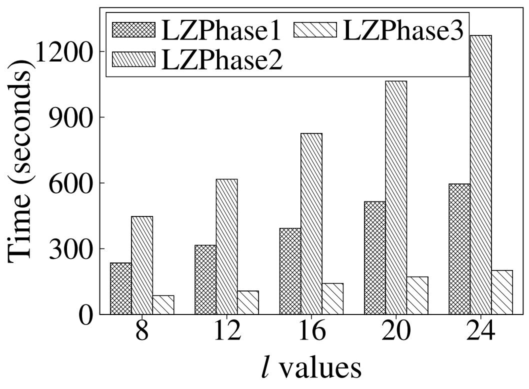
(b) Running time.



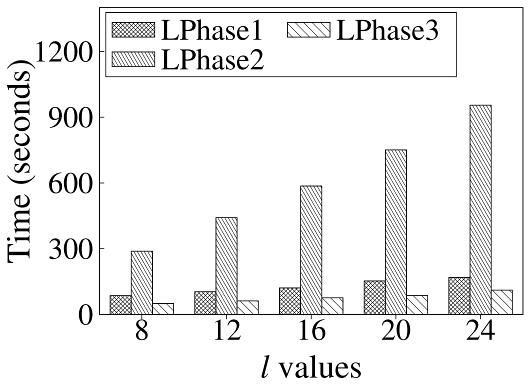
(c) Communication (GB).

Figure 4.9: Phase breakdown, running time, and communication vs $\alpha$ in (1x1) SIFT.

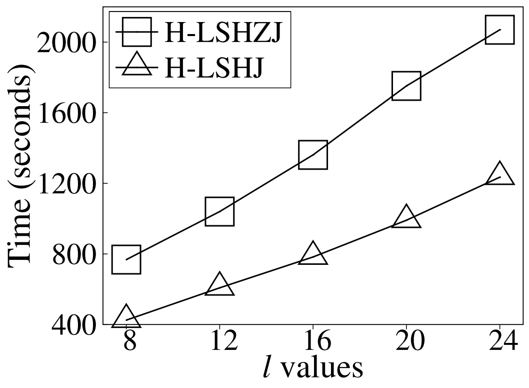
of buckets generated in H-LSHJ is inversely proportional to $w$ while the size of bucket is proportional to $w$. Initially, the running time of H-LSHJ is high because many buckets with



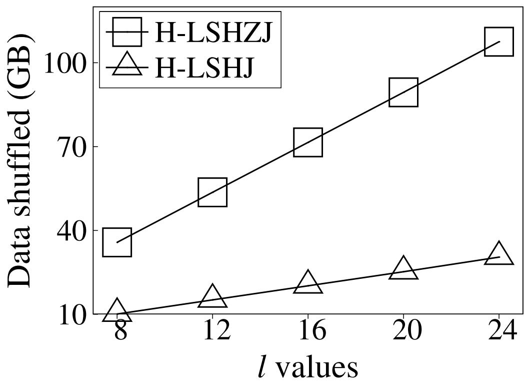
(a) H-LSHZJ breakdown.



(b) H-LSHJ breakdown.

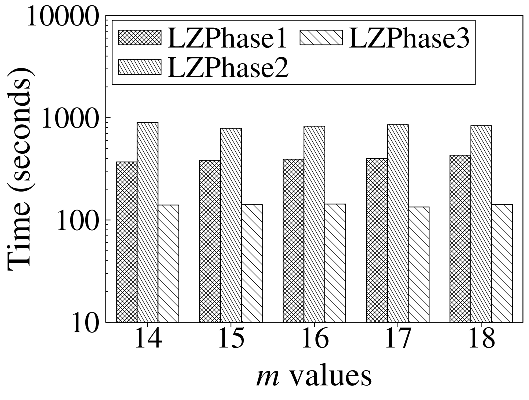


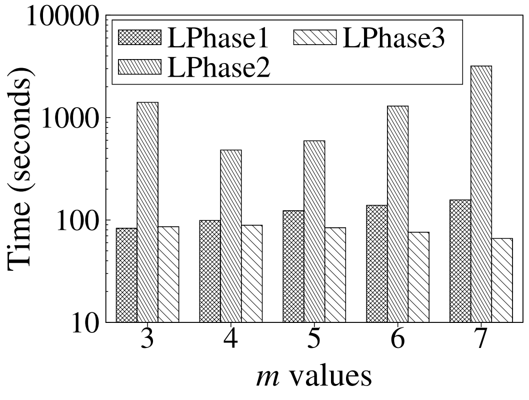
(c) Running time.



(d) Communication (GB).

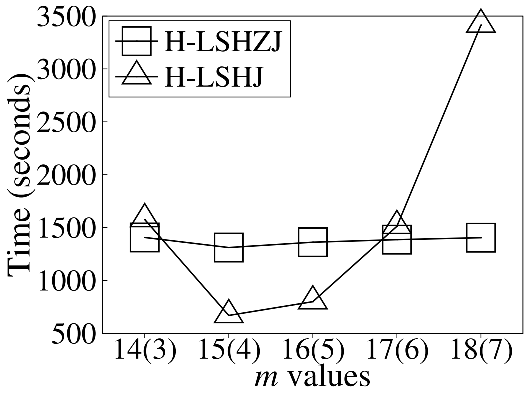
Figure 4.10: Phase breakdown, running time, and communication vs $l$ in (1x1) SIFT.

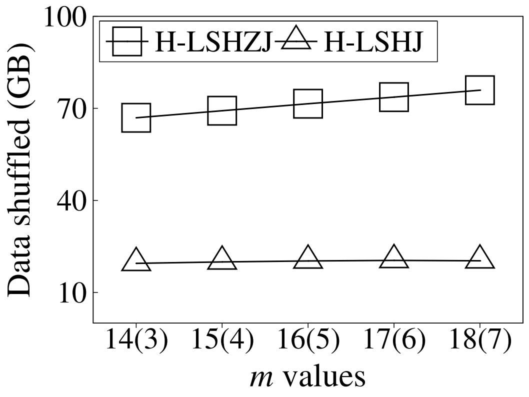




(a) H-LSHZJ breakdown.

(b) H-LSHJ breakdown.

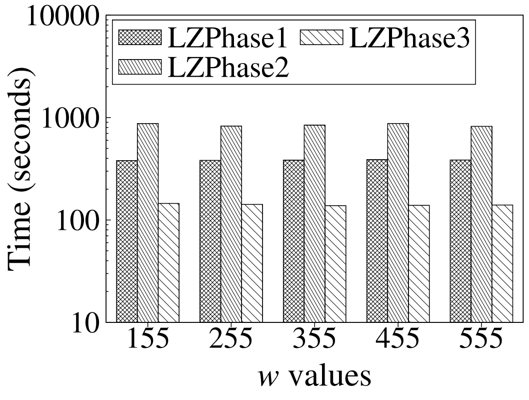




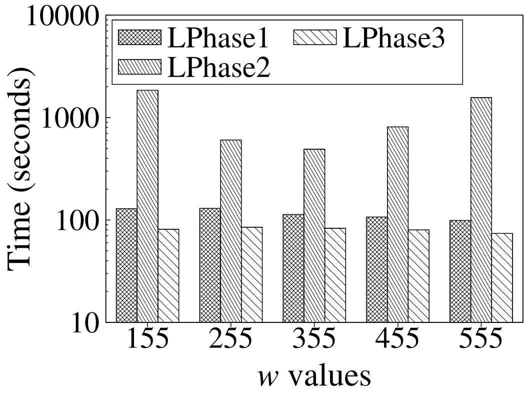
(c) Running time.

(d) Communication (GB).

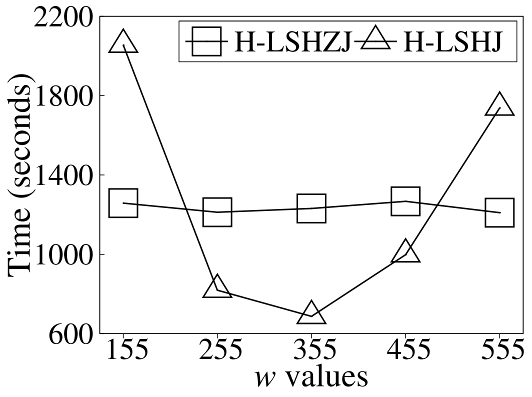
Figure 4.11: Phase breakdown, running time, and communication vs $m$ in (1x1) SIFT.



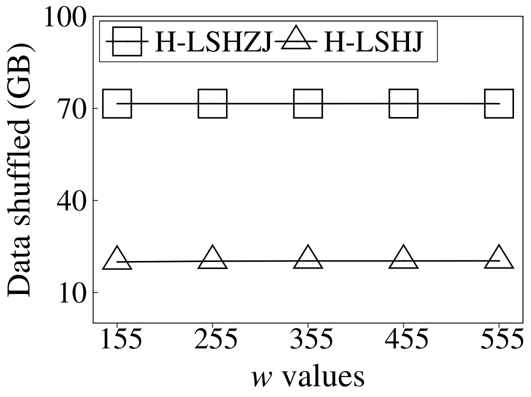
(a) H-LSHZJ breakdown.



(b) H-LSHJ breakdown.



(c) Running time.



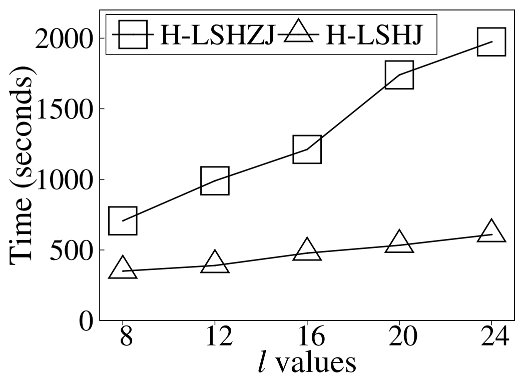
(d) Communication (GB).

Figure 4.12: Phase breakdown, running time, and communication vs $w$ in (1x1) SIFT.

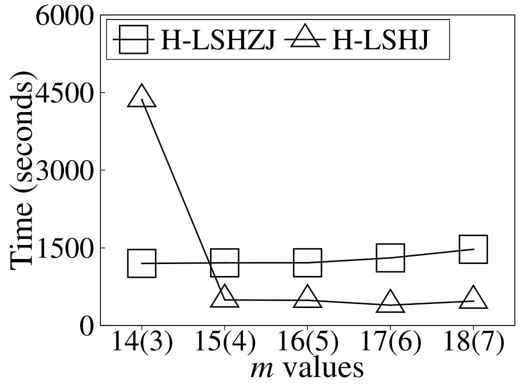
small amount of points incur lots of overhead, then its running time deceases as the number of buckets become less, until $w$ grows large enough such that only several big size buckets are created, which take a huge amount of time to process. The communication overhead of H-LSHJ and H-LSHZJ are not affected a lot by different $w$s as indicated in Figure 4.12(d).

**Effect of $\alpha$, the random shift.** Figures 4.9(a) to 4.9(c) present the performance results of H-LSHZJ with different $\alpha$ (the random shift value). For H-LSHZJ algorithm, it needs to transform high-dimensional datasets into $\alpha l$ one-dimensional datasets, therefore, all three stages have to process more data with increased $\alpha$, which explains the running times of LZPhase1, LZPhase2, and LZPhase3 all increase linearly as the value of $\alpha$ grows illustrated by Figures 4.9(a) to 4.9(b). Similarly, the communication cost given by Figure 4.9(c) also increases linearly with $\alpha$.

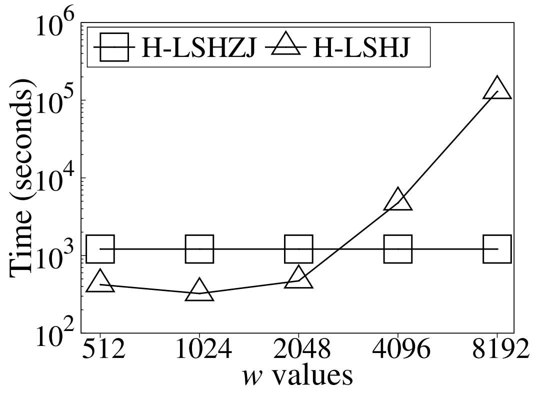
**Effect of Skewed Data** Figure 4.13 to Figure 4.14 present the performance results of H-LSHZJ and H-LSHJ using (1x1) Cluster datasets. It is obvious from Figure 4.14(c) and Figure 4.13(c) that H-LSHZJ delivers much better approximation quality than H-LSHJ does while the running time time for H-LSHZJ is orders of magnitude better that of H-LSHJ. Figure 4.14(c) indicates that the approximation ratio for H-LSHZJ is round 1.02 and is never beyond 1.05 even in the worst case. H-LSHJ cannot return the complete $k$NN join results even when we increase $w$ to 8096. To test the how $l$, $m$, and $w$ affect H-LSHJ, we fix $w$ to 2048 to ensure H-LSHJ can finish execution in reasonable time. As for H-LSHZJ, we set $l$ as 16, $m$ as 16, and $w$ as 255 for all default cases. Figure 4.13(a) and 4.14(a) show that the running times of both H-LSHJ and H-LSHZJ keep increasing as $l$ grows, and at the same time, H-LSHZJ's approximation ratio and the missing records of H-LSHJ become less. Different $m$ values has little effect on H-LSHZJ as illustrated by Figure 4.13(b) and Figure 4.14(b). If $m$ is small, for example when $m$ is 3, the running time of H-LSHJ is high because it has to spend substantial amount of time dealing with large buckets. With larger $m$, H-LSHJ's running time becomes less since the number of points contained in every bucket gradually reduced as in Figure 4.13(b) and Figure 4.14(b). Clearly, varying $w$ has



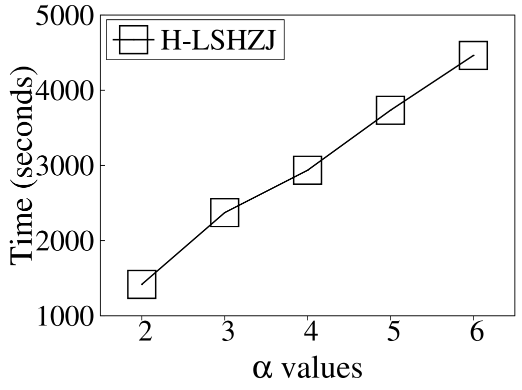
(a) Vary $l$.



(b) Vary $m$.

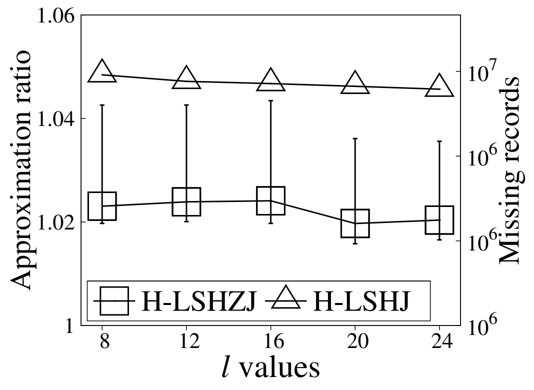


(c) Vary $w$.

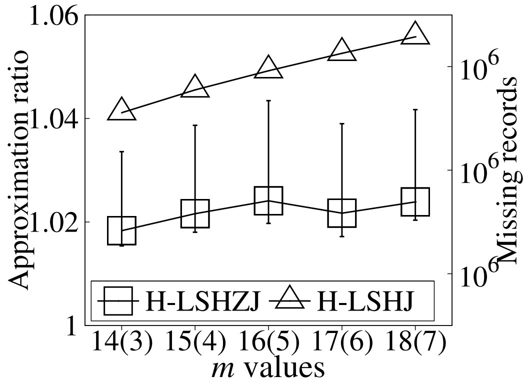


(d) Vary $\alpha$.

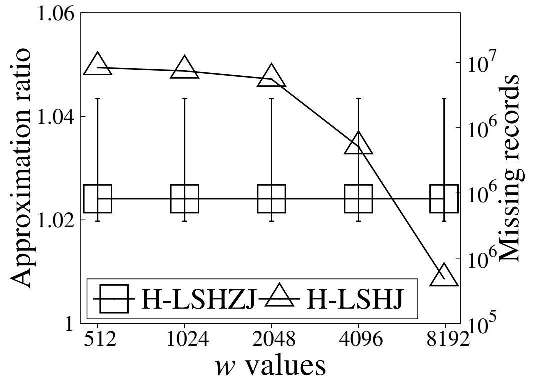
Figure 4.13: Running time of H-LSHJ and H-LSHZJ on Cluster.



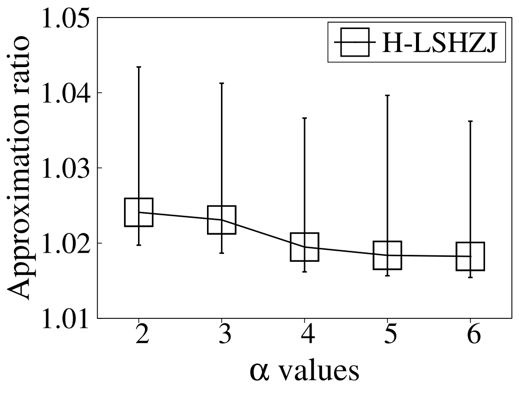
(a) Missing results with varied $l$.



(b) Missing results with varied $m$.



(c) Missing results with varied $w$.



(d) Vary $\alpha$.

Figure 4.14: Approximation quality of H-LSHJ and H-LSHZJ on Cluster.

no effect on the performance of H-HSLJ as it can be seen from 4.13(c) and 4.14(c). As for H-HSLJ, larger $w$ means larger buckets, leading to the increase of its running time and the reduction of its missing records. Similar to the $\alpha$'s effect on SIFT datasets, increasing $\alpha$ could deliver better approximation quality of H-LSHZJ, but it would demand more running time from the algorithm, which are verified by Figure 4.13(d) and Figure 4.14(d).

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

## 5.1 Conclusion

In this dissertation, we investigate the challenges in parallel similarity joins. In particular, we focused on the $k$NN join problem in a parallel and distributed environment. We proposed both exact and approximate solutions to $k$NN join in both low and high dimensions using MapReduce.

In Chapter 2, we formalize the notion of $k$NN join and present the related works on topics such as $k$NN queries, $k$NN joins, and $k$NN graph. We also discuss existing exact and approximate $k$NN joins algorithms in both sequential centralized and parallel environments.

In Chapter 3, we give our motivation to design $k$NN join algorithm for low-dimensional data set using the popular MapReduce parallel programming paradigm. We first propose a block nested loop join based solution, then improve the algorithm by building R-tree indices. Due to the limitations (excessive communication and computation overheads) of the exact solutions, we presents an efficient approximate solution (based on $z$-order curves), which transform $k$NN joins into a number of range queries on $B^+$-trees built upon $z$-order values. In this process, we have to address a number of interesting challenges related with load balancing and scalability issues. This solution ensures the good performance of the join operation, and at the same time, guarantees high quality of the (approximate) join results.

In Chapter 4, we investigate how to perform efficient $k$NN join for high-dimensional large data sets using MapReduce. Due to the curse of dimensionality, it is prohibitively expensive to compute exact solutions for high-dimensional data, and space filling curves also become ineffective and expensive to use in high dimensions. Therefore, we resort to approximate approaches based on the popular LSH techniques which are widely used in processing similarity search in high dimensions. The main idea of the algorithm works as follows. First, it reduces high-dimensional data into low-dimensional data; second, it maps low-dimensional data into one-dimensional $z$-order values. Lastly, it performs efficient range queries on $z$-order values and returns the final join results. We also address interesting challenges related with load balancing and scalability issues, when realizing this idea in MapReduce. We conduct extensive experiments on high-dimensional large data sets to verify the performance and approximation quality of our proposed parallel $k$NN join algorithms.

## 5.2 The Future Work

Our study in this dissertation focuses on the popular parallel and distributed programming paradigm MapReduce. There are several other parallel popular parallel computing paradigms, for instance, MPI and Spark [1] (which is a recently-proposed in-memory based computing model on a cluster of commodity machines). Extending our work to these other popular parallel programming frameworks presents some interesting challenges. For example, to implement a parallel $k$NN join solution using MPI, we could use a similar idea in [40] to extend our results, which distributes the workload among $n$ computing nodes. and communicates only required information among computing nodes.

The MapReduce framework is very suitable for batched, offline processing of write-once and read-many data sets. However, MapReduce has its limitations in supporting iterative workloads. To implement iterative algorithms using MapReduce, one has to start several rounds of MapReduce, which could lead to considerable overhead because data has to be materialized back to disks at the end of each map stage and reduce stage and read from disks in the next stage. Spark provides primitives for in-memory cluster computing and applications can load data into memory and query and process it repeatedly using a particular construction called RDD (resilient distributed dataset), which is much faster than disk-based systems like Hadoop. Therefore, it is an interesting open problem to design iterative, parallel methods to solve $k$NN joins more efficiently in these frameworks. For example, $k$NN can be solved by a number of distance range queries if we keep increasing the search radius until enough number of nearest neighbors are found, therefore, we could extend the idea proposed by [44] to design iterative-based parallel exact $k$NN join algorithms for high-dimensional data sets using Spark.

Lastly, this dissertation focused on parallel $k$NN joins in euclidean space. Most of our results still apply to parallel similarity joins using a different similarity/distance function, as long as it is in a metric space, by using embedding techniques. However, working out the details and extending our results to non-metric space represent interesting challenges to be addressed in future work.

# BIBLIOGRAPHY

[1] Apache spark - lightning-fast cluster computing. `http://spark.incubator.apache.org`.

[2] Open street map. `http://www.openstreetmap.org`.

[3] The openmp api specification for parallel programming. `http://openmp.org/wp`.

[4] Afsin Akdogan, Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. Voronoi-based geospatial query processing with mapreduce. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 9–16, Washington, DC, USA, 2010. IEEE Computer Society.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of ACM*, 45(6):891–923, 1998.

[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[7] R.E. Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 2003.

[8] Richard E. Bellman. *Adaptive control processes - A guided tour*. Princeton University Press, Princeton, New Jersey, U.S.A., 1961.

[9] Stefan Berchtold and Christian Böhm. A cost model for nearest neighbor search in high-dimensional data space. In *In Proc. ACM Symp. on Principles of Database Systems*, 1997.

[10] Christian Böhm and Florian Krebs. Supporting kdd applications by the k-nearest neighbor join. In *In Proc. of DEXA*, pages 504–516. Springer, 2003.

[11] Christian Böhm and Florian Krebs. The k-nearest neighbor join: Turbo charging the kdd process. *Knowledge and Information Systems (KAIS)*, 6:728–749, 2004.

[12] Christian Böhm and Hans-Peter Kriegel. A cost model and index architecture for the similarity join. In *ICDE*, 2001.

[13] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Parallel processing of spatial joins using r-trees. In *ICDE*, 1996.

[14] K. Selçuk Candan, Parth Nagarkar, Mithila Nagendra, and Renwei Yu. RanKloud: a scalable ranked query processing framework on hadoop. In *EDBT*, 2011.

[15] T. M. Chan. Approximate nearest neighbor queries revisited. In *SoCG*, 1997.

[16] Damien François, Vincent Wertz, and Michel Verleysen. The concentration of fractional distances. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):873–886, 2007.

[17] M. Connor and P. Kumar. Parallel construction of k-nearest neighbor graphs for point clouds. In *Eurographics Symposium on PBG*, 2008.

[18] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Trans. Vis. Comput. Graph.*, 16(4):599–608, 2010.

[19] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 2004.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[21] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[22] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[23] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[24] Hadoop Project. http://hadoop.apache.org/.

[25] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[26] E.G. Hoel and H. Samet. Data-parallel spatial join algorithms. In *ICPP*, 1994.

[27] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[28] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV*, 2008.

[29] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, 1990.

[30] Jon M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 599–608, New York, NY, USA, 1997. ACM.

[31] N. Kouiroukidis and G. Evangelidis. The effects of dimensionality curse in high dimensional knn search. In *Informatics (PCI), 2011 15th Panhellenic Conference on*, pages 41–45, 2011.

[32] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD*, 2011.

[33] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[34] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *VLDB*, 2012.

[35] Gang Luo, J.F. Naughton, and C.J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, 2002.

[36] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[37] L. Mutenda and M. Kitsuregawa. Parallel r-tree spatial join for a shared-nothing architecture. In *DANTE*, 1999.

[38] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

[39] Jignesh M. Patel and David J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *ACM GIS*, 2000.

[40] Erion Plaku and Lydia E. Kavraki. Distributed computation of the kNN graph for large high-dimensional point sets. *J. Parallel Distrib. Comput.*, 67(3):346–359, 2007.

[41] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[42] Donovan A. Schneider and David J. Dewitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. pages 110–121, 1989.

[43] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning And Vision: Theory And Practice*. ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS. Mit Press, 2005.

[44] Vishwakarma Singh and Ambuj K. Singh. Simp: accurate and efficient near neighbor search in high dimensional spaces. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 492–503, New York, NY, USA, 2012. ACM.

[45] Aleksandar Stupar, Sebastian Michel, and Ralf Schenkel. RankReduce - processing K-Nearest Neighbor queries on top of MapReduce. In *LSDS-IR*, 2010.

[46] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, 2009.

[47] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[48] Dilin Wang, Yanmei Zheng, and Jianwen Cao. Parallel construction of approximate knn graph. *2013 12th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*, 0:22–26, 2012.

[49] Jijie Wang, Lei Lin, Ting Huang, Jingjing Wang, and Zengyou He. Efficient k-nearest neighbor join algorithms for high dimensional sparse data. *CoRR*, abs/1011.2807, 2010.

[50] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.

[51] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[52] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient b-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

[53] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jing Hu. Gorder: an efficient method for knn join processing. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 756–767. VLDB Endowment, 2004.

[54] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE*, 2010.

[55] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based knn join processing for high-dimensional data. *Information & Software Technology*, 49(4):332–344, 2007.

[56] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[57] Cui Yu, Rui Zhang, Yaochun Huang, and Hui Xiong. High-dimensional knn joins with incremental updates. *Geoinformatica*, 14(1):55–82, 2010.

[58] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, 2012.

[59] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial queries evaluation with mapreduce. In *GCC*, 2009.

[60] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*, 2009.

[61] Xiaofang Zhou, David J. Abel, and David Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2:175–204, 1998.

# BIOGRAPHICAL SKETCH

Chi Zhang joined the Florida State University in the fall of 2006 to pursue his PhD in the Department of Computer Science. His research interests include databases and data management, distributed and parallel query processing, and high performance computing.