Efficient Parallel kNN Joins for Large Data in MapReduce

Chi Zhang[1] Feifei Li[2] Jeffrey Jestes[2]

[1]Computer Science Department
Florida State University
czhang@cs.fsu.edu

[2]School of Computing
University of Utah
{lifeifei, jestes}@cs.utah.edu

ABSTRACT

In data mining applications and spatial and multimedia databases, a useful tool is the kNN join, which is to produce the k nearest neighbors (NN), from a dataset S, of every point in a dataset R. Since it involves both the join and the NN search, performing kNN joins efficiently is a challenging task. Meanwhile, applications continue to witness a quick (exponential in some cases) increase in the amount of data to be processed. A popular model nowadays for large-scale data processing is the shared-nothing cluster on a number of commodity machines using *MapReduce* [6]. Hence, how to execute kNN joins efficiently on large data that are stored in a MapReduce cluster is an intriguing problem that meets many practical needs. This work proposes novel (exact and approximate) algorithms in MapReduce to perform efficient parallel kNN joins on large data. We demonstrate our ideas using Hadoop. Extensive experiments in large real and synthetic datasets, with tens or hundreds of millions of records in both R and S and up to 30 dimensions, have demonstrated the efficiency, effectiveness, and scalability of our methods.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems. Subject: Query processing*

General Terms

Algorithms

1. INTRODUCTION

The k-nearest neighbor join (kNN join) is an important and frequently used operation for numerous applications including knowledge discovery, data mining, and spatial databases [2,14,20,22]. Since both the join and the nearest neighbor (NN) search are expensive, especially on large data sets and/or in multi-dimensions, kNN-join is a costly operation. Lots of research have been devoted to improve the performance of kNN joins by proposing efficient algorithms [20,22].

However, all these approaches focus on methods that are to be executed in a single thread on a single machine. 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 [8,15], set similarity joins [17], relational θ-joins [12], and spatial range joins defined by distance threshold [25] in MapReduce have been designed and implemented, there has been little work on parallel kNN joins in large data, which is a challenging task and becoming increasingly essential as data sets continue to grow at an exponential rate.

When dealing with extreme-scale data, parallel and distributed computing using shared-nothing clusters, which typically consist of a number of commodity machines, is quickly becoming a dominating trend. MapReduce [6] was introduced with the goal of providing a simple yet powerful parallel and distributed computing paradigm. The MapReduce architecture also provides good scalability and fault tolerance mechanisms. 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.

Motivated by these observations, this work investigates the problem of executing kNN joins for large data using MapReduce. We first propose the basic approach using block-nested-loop-join (BNLJ) and its improved version using the R-tree indices. 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 multi-dimensional data. In light of its limitation, we introduce a MapReduce-friendly, approximate algorithm that is based on mapping multi-dimensional data sets into one dimension using space-filling curves (z-values), and transforming kNN 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.

We present the above algorithms in MapReduce and 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 30 dimensions) demonstrate that our approximate method consistently outperforms the basic approach by at least 1-2 orders of magnitude, while achieving very good approximation quality.

The paper is organized as follows. Section 2 introduces background information. Section 3 discusses baseline methods and Section 4 presents our parallel kNN join algorithms in MapReduce. Section 5 reports experimental results. Section 6 discusses related work and Section 7 concludes.

2. BACKGROUND

2.1 kNN 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, $\text{knn}(r, S)$ returns the set of k nearest neighbors (kNN) of r from S, where ties are broken arbitrarily. People are also often interested in finding just the approximate k nearest neighbors. Let $\text{aknn}(r, S)$ be a set of approximate k nearest neighbors for r in S. Assume r's exact kth nearest neighbor in $\text{knn}(r, S)$ is point p. Let p' be the kth nearest neighbor in $\text{aknn}(r, S)$. Then, we say $\text{aknn}(r, S)$ is a *c-approximation* of $\text{knn}(r, S)$ for some constant c if and only if:

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

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

kNN join: The kNN join $\text{knnJ}(R, S)$ of R and S is:

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

Approximate kNN join: The approximate kNN join of R and S is denoted as $\text{aknnJ}(R, S)$ and is expressed as:

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

2.2 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 jth ($1 \leq j \leq m$) 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 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. BASELINE METHODS

The straightforward method for kNN-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 kNN join between the local R and S blocks in that bucket, i.e., find kNNs 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 kNNs 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 kNNs of r w.r.t. the corresponding block from S. Hence, in the second MapReduce phase, our job is to find the global kNNs for every record $r \in R$ among its n local

kNNs 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 kNNs 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 kNNs of a record r from the local R block in the same bucket. Specifically, for each block S_{bj} ($1 \le j \le n$), we build a reducer-local spatial index over S_{bj}, in particular we used the R-tree, before proceeding to find the local kNNs for every record from the local R block in the same bucket with S_{bj}. Then, we use kNN functionality from R-tree to answer $\mathrm{knn}(r, S_{bj})$ in every bucket in the reducer. Bulk-loading a R-tree for S_{bj} is very efficient, and kNN 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 kNNs 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 kNNs 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 kNN 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 kNNs 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 pre-processing 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 [18, 19].

4. Z-VALUE BASED PARTITION JOIN

In *H-BNLJ* and *H-BRJ*, for each $r \in R$ we must compute $\mathrm{knn}(r, S_{bj})$ 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 [21] 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 [21]] 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 $\{\mathbf{v}_2, \dots, \mathbf{v}_\alpha\}$, such that for any i, $\mathbf{v}_i \in \mathbb{R}^d$, and shift P by these vectors to obtain $\{P_1, \dots, P_\alpha\}$ ($P_1 = P$). Then, Algorithm 1 returns a constant approximation in any fixed dimension for* $\mathrm{knn}(q, P)$ *in expectation.*

Algorithm 1: zkNN(q, P, k, α)   [from [21]]

```
1 generate $\{\mathbf{v}_2, \dots, \mathbf{v}_\alpha\}$, $\mathbf{v}_1 = \overrightarrow{0}$, $\mathbf{v}_i$ 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 $i = 1, \dots, \alpha$ 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 (1)
5     insert $z^+(z_{q_i}, k, P_i)$ into $C_i(q)$;    // see (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 $\mathrm{knn}(q, C(q))$.
```

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

$$z^-(z_p, k, P) = k \text{ points immediately preceding } z_p \text{ in } \mathrm{Z}_P \quad (1)$$

$$z^+(z_p, k, P) = k \text{ points immediately succeeding } z_p \text{ in } \mathrm{Z}_P \quad (2)$$

The idea behind zkNN is illustrated in Figure 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 $\{C_1(q), \dots, C_\alpha(q)\}$, and the overall candidate set $C(q) = \bigcup_{i=1}^{\alpha} C_i(q)$. As seen in Figure 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 (1) (or (2)) near the head (tail) of 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.

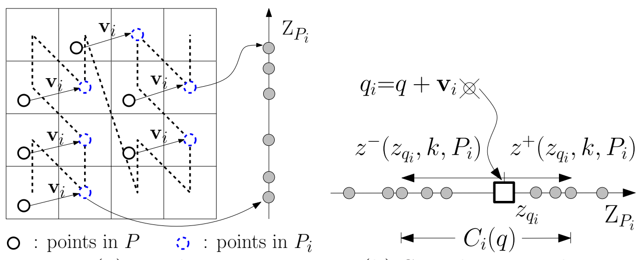


Figure 1: Algorithm zkNN.

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 [21]). However, to efficiently apply the above idea for kNN 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 kNN-joins in MapReduce by leveraging the idea in zkNN, dubbed *H-zkNNJ* (Hadoop based zkNN Join).

4.1 zkNN Join in MapReduce

Overview of H-zkNNJ. We generate α 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 α 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 kNN, according to zkNN, is in a small range ($2k$ points) surrounding z_r in Z_{S_i} (see Figure 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:

$$R_{i,j}, S_{i,j} = [z_{i,j-1}, z_{i,j}) \text{ (let } z_{i,0} = 0,\ z_{i,n} = +\infty); \quad (3)$$

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 2.

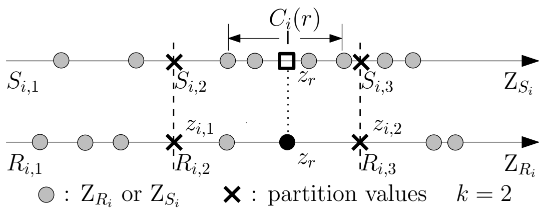


Figure 2: $C_i(r) \subseteq S_{i,j}$.

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 preceding (succeeding) points are met. An example is shown in Figure 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 4.

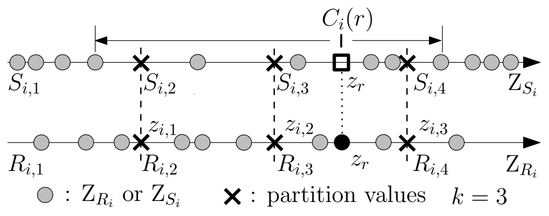


Figure 3: $C_i(r) \not\subseteq S_{i,j}$.

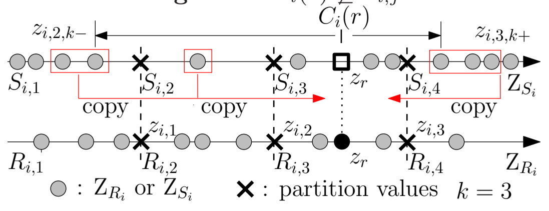


Figure 4: Copy $z^-(z_{i,j-1}, k, S_i), z^+(z_{i,j}, k, S_i)$ **to** $S_{i,j}$.

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 (1), (2), and (3), $z^-(z_{i,j-1}, k, S_i)$ and $z^+(z_{i,j}, k, S_i)$ are copied into $S_{i,j}$, i.e.,

$$S_{i,j} = [z_{i,j-1}, z_{i,j}) \cup z^-(z_{i,j-1}, k, S_i) \cup z^+(z_{i,j}, k, S_i). \quad (4)$$

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 (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 jth 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 2.2, 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 jth 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 jth 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 jth block $S_{i,j}$. In this case, the jth 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 (4)). The bottleneck of the Reduce phase is apparently the jth 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 jth block $R_{i,j}$. In this case, the jth 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 jth 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 ϕ-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 $B(r(x), p)$:

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

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

□

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:

$$x = \underset{x \in \widehat{D}}{\operatorname{argmin}}\, |\widehat{r}(x) - r|. \tag{5}$$

Lemma 2 *Any x returned by Equation (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 εN. When this happens, it means that no elements with ranks that are within εN 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:

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

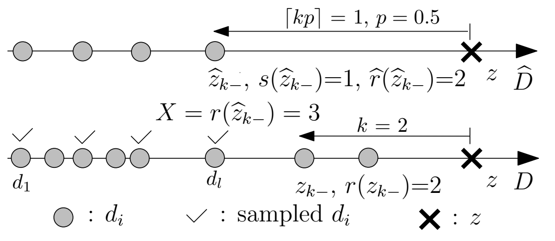
□

Thus, we can use $\widehat{D}$ and (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 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 Z_{R_i}. In order to partition S_i using these z-values, we need to ensure that, in the jth 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 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 kth 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 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 kth 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 kth 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 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 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.

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



$\lceil kp \rceil = 1,\ p = 0.5$
$\widehat{z}_{k-},\ s(\widehat{z}_{k-})$=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 5: Estimate z_{k-} given any value z.

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 $a \in [\lceil kp \rceil, r(d_1)]$,*

$$\Pr[X = a] = p \binom{a-1}{\lceil kp \rceil - 1} p^{\lceil kp \rceil - 1}(1-p)^{a - \lceil kp \rceil}, \quad (6)$$

$$\textit{and } \Pr[X - k] \le \varepsilon N \ge 1 - e^{-2/\varepsilon} \textit{ if } p = 1/(\varepsilon^2 N). \quad (7)$$

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

- e_1: the ath element d_ℓ ($r(d_\ell) = a$) 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_ℓ 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 (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 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 kth 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, $\Pr[X - k] \le \varepsilon N \ge 1 - e^{-2/\varepsilon}$. □

Theorem 3 implies that boundary values $z_{i,j-1,k-}$ and $z_{i,j,k+}$ for any jth 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 Z_{R_i}).

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

4.2 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 $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_\alpha\}$, 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

Algorithm 2: H-zkNNJ(R, S, k, n, ε, α)

```
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 $(r, \mathrm{knn}(r, C(r))$;
```

Algorithm 3: estimator1($\widehat{R}$, ε, 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 $\mathrm{Z}_{\widehat{S}}$.
2  return the $\lceil kp \rceil$th value to the left and the $\lceil kp \rceil$th
   value to the right of $z$ in $\mathrm{Z}_{\widehat{S}}$.
```

which partitions each emitted record into one of α partitions by the shift identifier i so all records from the ith 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 α 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 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 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 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 6.

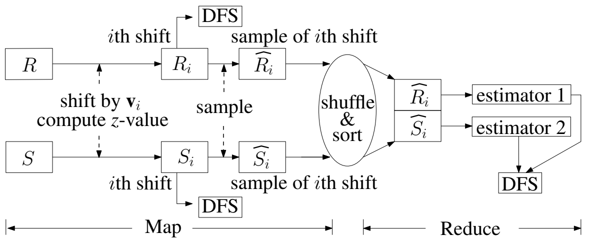


Figure 6: H-zkNNJ MapReduce Phase 1

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 $\mathrm{knn}(r, S)$ 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 αn 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 ℓ 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 αn partitions based on ℓ.

The reducers compute $C_i(r)$ for each $r \in R_{i,j}$ in a partition $(R_{i,j}, S_{i,j})$; αn reducers are started, each handling one of the αn $(R_{i,j}, S_{i,j})$ 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 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 $s \in \mathrm{knn}(r, C_i(r))$. There is one such file per reducer. The second MapReduce phase is illustrated in Figure 7.

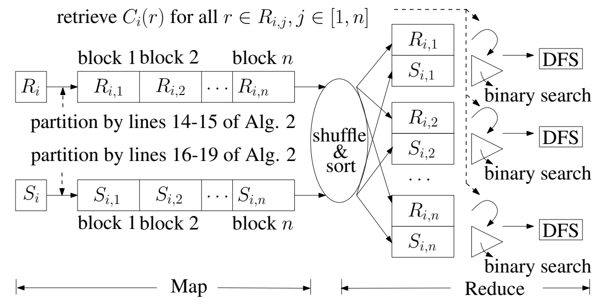


Figure 7: H-zkNNJ MapReduce Phase 2

Phase 3: The last phase decides $\mathrm{knn}(r, C(r))$ for any $r \in R$ from the $\mathrm{knn}(r, C_i(r))$'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 ε values we are copying roughly $O(k)$ values only for each block. For αn 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 $\mathrm{knn}(r, C_i(r))$ for each $r \in R_i$ giving us a total of αk 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 α 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 α 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 αn 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 α 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 kNN 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.

5. EXPERIMENTS

5.1 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. 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.

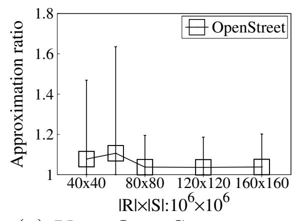
A record's z-value is always stored and communicated as a byte array. To test our algorithms' performance, we analyze end-to-end running time, communication cost, and speedup and scalability. We generate a number of different datasets as R and S from the complete OpenStreet dataset (50 states) by randomly 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 (40×80) dataset has 40 million R and 80 million S records. Unless otherwise noted (40×40) 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 γ be the number of physical slave machines used 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 αn 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, $\gamma = 16$, $k = 10$.

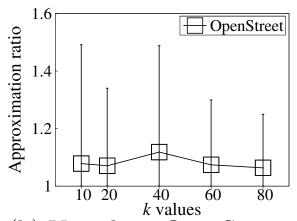
5.2 Performance Evaluation

Approximation quality. For our first experiment we analyze the approximation quality of H-zkNNJ. Since obtaining the exact kNN 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 kth-NN (returned by the H-zkNNJ) and its distance to the exact kth-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 8(a) to 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 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 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 dataset sizes in Figure 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 8(d). We also extensively analyze the effect dimensionality d and the number of shifts α have on the approximation quality in later experiments.

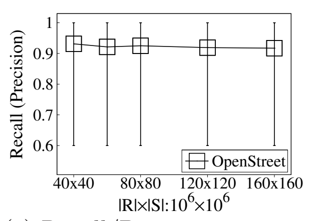
Effect of ε. Next we analyze the effect ε has on the performance of H-zkNNJ using a (40x40) OpenStreet dataset. We notice in Figure 9(a) the running time of H-zkNNJ decreases as we vary ε from 0.1 to 0.003, due to better load-balancing resulted from more equally distributed block sizes for $R_{i,j}$ and $S_{i,j}$. Decreasing ε 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



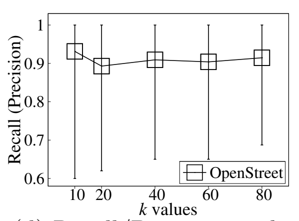
(a) Vary OpenStreet size.



(b) Vary k on OpenStreet.

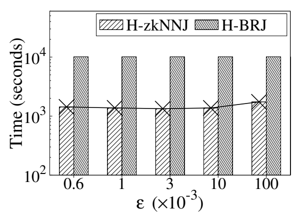


(c) Recall/Precision vs size.

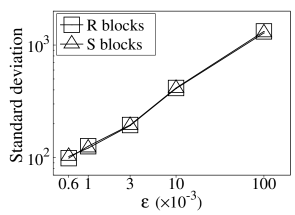


(d) Recall/Precision vs k.

Figure 8: Approximation quality of H-zkNNJ on (40x40) OpenStreet.

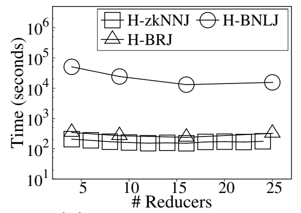


(a) Running time.

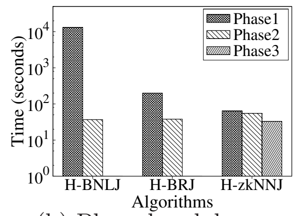


(b) Block size SD.

Figure 9: Effect of ε in H-zkNNJ on (40x40).



(a) Running time.



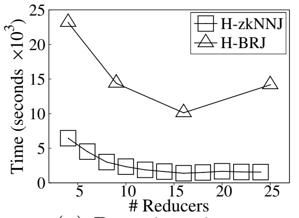
(b) Phase breakdown.

Figure 10: Running time for (1x1) OpenStreet.

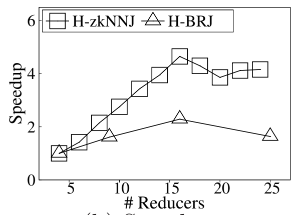
more balanced partitions in the second MapReduce phase. Regardless of the value of ε, H-zkNNJ is always an order of magnitude faster than H-BRJ. Figure 9(b) analyzes the standard deviation (SD) in the number of records contained in the $R_{i,j}$ and $S_{i,j}$ blocks in the reduce phase of the second 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 5×10^6 records to achieve the optimal load balance. Figure 9(b) shows our sampling-based partition method is highly effective, achieving a SD from 100 to 5000 when ε changes from 0.0006 to 0.1. There is a steady decrease in SD for blocks created from R and S as we decrease ε (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 4×10^7; $\varepsilon = 0.003$ 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 γ (the number of physical machines running as slaves in the Hadoop cluster) are shown in Figure 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 γ. 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 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. The time breakdown of different phases of all algorithms with $r = \gamma = 16$ is depicted in Figure 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.



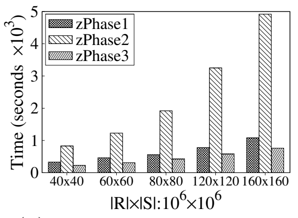
(a) Running time.



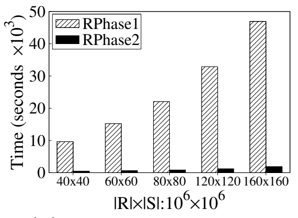
(b) Speedup.

Figure 11: Running time and speedup in (40x40).

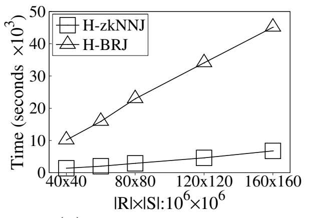
Using the same parameters for n, r, γ as above, we use a (40x40) OpenStreet dataset to further analyze the speedup and running times of H-zkNNJ and H-BRJ in varied cluster sizes. Figure 11(a) shows both running times reduce as the number of physical nodes increases. But as the number of reducers exceeds 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 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



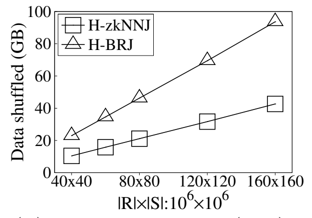
(a) H-zkNNJ breakdown.



(b) H-BRJ breakdown.

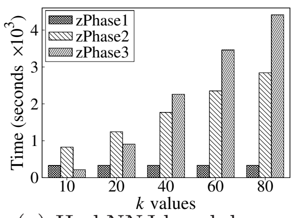


(c) Running time.

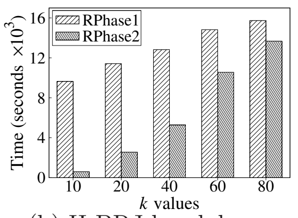


(d) Communication (GB).

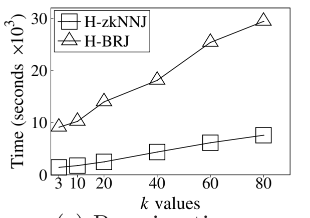
Figure 12: Phase breakdown, running time, and communication vs $|R| \times |S|$.



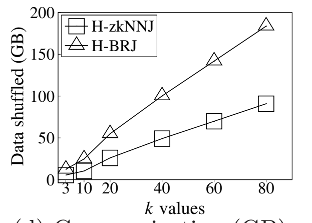
(a) H-zkNNJ breakdown.



(b) H-BRJ breakdown.



(c) Running time.



(d) Communication (GB).

Figure 13: Phase breakdown, running time, and communication vs k in (40x40) OpenStreet.

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 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 the speedup of H-zkNNJ is $r/(2\sqrt{r})$ times more, which agrees with the trend in Figure 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 12(a) and 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.

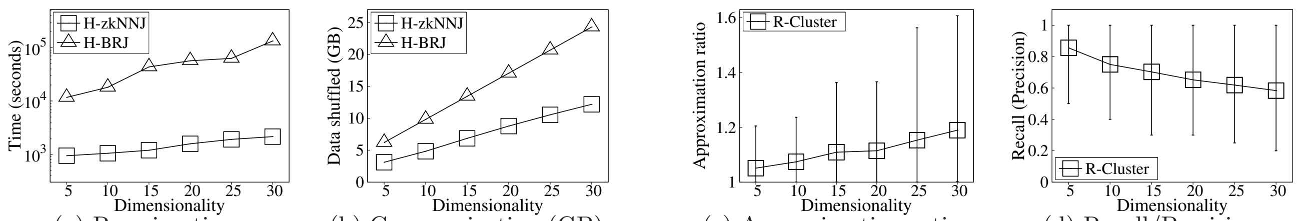
Clearly H-zkNNJ delivers much better running time performance than H-BRJ from Figure 12(c). The performance of H-zkNNJ is at least an order of magnitude better than H-BRJ when dealing with large datasets. The trends in Figure 12(c) also indicate H-zkNNJ becomes increasingly more efficient than H-BRJ 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 αn partitions; (2) Given the same number of blocks n in R and S, H-zkNNJ requires fewer machines to process all partitions for the kNN-join of R and S in parallel than H-BRJ does. H-zkNNJ needs αn 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 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 12(c). It takes less than 100 minutes completing kNN 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, we use (40x40) by default.

Effect of k. Figures 13(a)–13(c) present running times for H-zkNNJ and H-BRJ with different k, using (40x40) OpenStreet datasets. Figure 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 13(b).

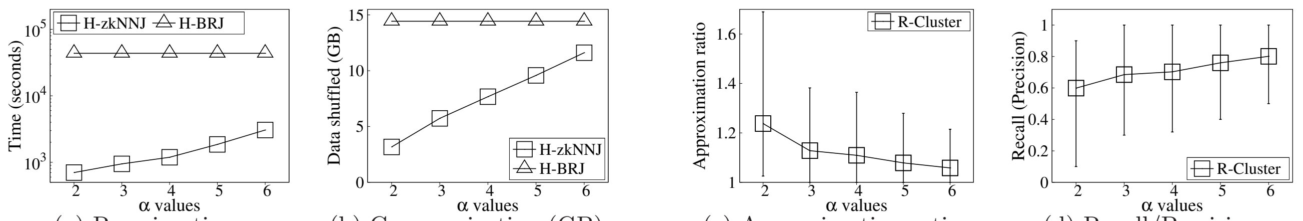
Figure 13(c) shows H-zkNNJ performs consistently (much) better than H-BRJ in all cases (from $k = 3$ to 80). For small k values, kNN join operations are the determining factors for performance. For example, the performance of H-zkNNJ given by Figure 13(a) is mainly decided by zPhase2 where k is 10. For 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 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.



(a) Running time. (b) Communication (GB). (c) Approximation ratio. (d) Recall/Precision.

Figure 14: Running time, communication, and approximation quality vs d in (5x5) R-Cluster, $\alpha = 3$.



(a) Running time. (b) Communication (GB). (c) Approximation ratio. (d) Recall/Precision.

Figure 15: Running time, communication, and approximation quality vs α in (5x5) R-Cluster, $d = 15$.

Figure 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 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 kNN join while H-BRJ needs more than 35 hours! In Figure 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 14(c) and 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 14(c). As we can see from 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 shifts $\alpha \in [2, 3, 4, 5, 6]$ and fix values of $d = 15$ and $k = 10$. Note that α shifts *imply* $(\alpha - 1)$ *random shifted copies plus the original dataset*.

Figures 15(a) and 15(b) indicate the performance time and communication cost of H-zkNNJ increase when we vary α from 2 to 6, due to the following reasons: (1) As α 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 α 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 15(c) and 15(d) show that both the approximation ratio and recall/precision of H-zkNNJ improve as α increases. The average approximation ratio decreases from 1.25 to below 1.1 and the average recall/precision increases from 60% to above 80%.

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 ε which determines how well balanced partitions are, hence the load balance of the system. Our experiments in Figure 9 show $\varepsilon = 0.003$ is already good enough to achieve a good load balance. Note ε 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 α which decides the approximation quality. By Theorem 1, using a small α returns a constant approximate solution on expectation and our experiments in Figures 8, 14(c), 14(d) and 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 γ are common system parameters for any MapReduce program which are determined based on resources in the cluster.

6. RELATED WORK

Performing kNN joins in the traditional setup has been extensively studied in the literature [2,20–23]. Nevertheless, these works focus on the centralized, single-thread setting that is not directly applicable in MapReduce. Early research on parallel join algorithms in a shared-nothing multiprocessor environment has been presented in [8,15], for relational join operators. Parallel spatial join algorithms aiming to join multiple spatial datasets according to a spatial join predicate (typically the intersection between two objects) using a multiprocessor system have been studied in [3,7,10, 11,13,26], none of which is designed for efficiently processing kNN-joins over distributively stored datasets. Recently, Zhang et al. [24,25] proposed a parallel spatial join algorithm in MapReduce, however, dealing with only spatial distance joins, which does not solve kNN-joins. An efficient parallel computation of the kNN graph over a dataset was designed using MPI (message passing interface) in [14]. Note the kNN graph is constructed over one data set, where each point is connected to its k nearest neighbors from the same dataset. As our focus is on kNN-joins over large datasets in MapReduce clusters, techniques for the MPI framework do not help solve our problem. Another parallel kNN graph construction method appears in [5], which works only for constructing the kNN graph over a centralized dataset. The goal there is to minimize cache misses for multi-core machines.

An efficient parallel set-similarity join in MapReduce was introduced in [17]; and the efficient processing of relational θ-joins in MapReduce was studied in [12]. The closest studies to our work appear in [1, 16] where kNN 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 kNN query algorithm, it is still quite challenging to design efficient MapReduce based kNN join algorithms (i.e., how to design partitioning to minimize communication, how to achieve load balancing). Furthermore, the LSH-based approach works for data in very high dimensions and the voronoi diagram based approach only works for data in 2-dimensions. In contrast, our goal is to design practical, efficient kNN 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 that 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) [18, 19], or possible solutions built on systems/frameworks that extend the standard MapReduce paradigm [4]. Join processing in MapReduce leveraging columnar storage was investigated in [9], which does require a new file format. Extending our study to this storage engine is an interesting future work.

7. CONCLUSION

This work studies parallel kNN joins in MapReduce. Exact (H-BRJ) and approximate algorithms (H-zkNNJ) are proposed. By delicately constructing partition functions for H-zkNNJ, we only require linear number of reducers (to the number of blocks from partitioning the base dataset), in contrast to the quadratic number of reducers required in the exact method. H-zkNNJ delivers performance which is orders of magnitude better than baseline methods, as evidenced from experiments on massive real datasets. It also achieves excellent quality in all tested scenarios. For future work, we plan to study kNN-joins in very high dimensions and other metric space in MapReduce.

8. ACKNOWLEDGMENT

Chi Zhang, Feifei Li, and Jeffrey Jestes were supported in part by NSF Grants IIS-1212310 and IIS-1200792.

9. REFERENCES

[1] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *CloudCom*, 2010.

[2] C. Böhm and F. Krebs. The k-nearest neighbor join: Turbo charging the kdd process. *KAIS*, 6:728–749, 2004.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *ICDE*, 1996.

[4] K. S. Candan, P. Nagarkar, M. Nagendra, and R. Yu. RanKloud: a scalable ranked query processing framework on hadoop. In *EDBT*, 2011.

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

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

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

[8] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer. In *VLDB*, 1990.

[9] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD*, 2011.

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

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

[12] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

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

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

[15] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*, 1989.

[16] A. Stupar, S. Michel, and R. Schenkel. RankReduce - processing K-Nearest Neighbor queries on top of MapReduce. In *LSDS-IR*, 2010.

[17] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[18] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.

[19] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

[20] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: an efficient method for knn join processing. In *VLDB*, 2004.

[21] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE*, 2010.

[22] C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based knn join processing for high-dimensional data. *Inf. Softw. Technol.*, 49(4):332–344, 2007.

[23] C. Yu, R. Zhang, Y. Huang, and H. Xiong. High-dimensional knn joins with incremental updates. *Geoinformatica*, 14(1):55–82, 2010.

[24] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *GCC*, 2009.

[25] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*, 2009.

[26] X. Zhou, D. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2:175–204, 1998.