K Nearest Neighbour Joins for Big Data on MapReduce: a Theoretical and Experimental Analysis

Ge Song[†], Justine Rochas[*], Lea El Beze[*], Fabrice Huet[*] and Frédéric Magoulès[†]
[*]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
[†]CentraleSupélec, Université Paris-Saclay, France

ge.song@ecp.fr, {justine.rochas, fabrice.huet}@unice.fr, lea.elbeze@gmail.com ,
frederic.magoules@hotmail.com

Abstract—Given a point p and a set of points S, the kNN operation finds the k closest points to p in S. It is a computational intensive task with a large range of applications such as knowledge discovery or data mining. However, as the volume and the dimension of data increase, only distributed approaches can perform such costly operation in a reasonable time. Recent works have focused on implementing efficient solutions using the MapReduce programming model because it is suitable for distributed large scale data processing. Although these works provide different solutions to the same problem, each one has particular constraints and properties. In this paper, we compare the different existing approaches for computing kNN on MapReduce, first theoretically, and then by performing an extensive experimental evaluation. To be able to compare solutions, we identify three generic steps for kNN computation on MapReduce: data pre-processing, data partitioning and computation. We then analyze each step from load balancing, accuracy and complexity aspects. Experiments in this paper use a variety of datasets, and analyze the impact of data volume, data dimension and the value of k from many perspectives like time and space complexity, and accuracy. The experimental part brings new advantages and shortcomings that are discussed for each algorithm. To the best of our knowledge, this is the first paper that compares kNN computing methods on MapReduce both theoretically and experimentally with the same setting. Overall, this paper can be used as a guide to tackle kNN-based practical problems in the context of big data.

Index Terms—kNN, MapReduce, Performance Evaluation

1 Introduction

GIVEN a set of query points R and a set of reference points S, a k nearest neighbor join (hereafter kNN join) is an operation which, for each point in R, discovers the k nearest neighbors in S.

It is frequently used as a classification or clustering method in machine learning or data mining. The primary application of a kNN join is k-nearest neighbor classification. Some data points are given for training, and some new unlabeled data is given for testing. The aim is to find the class label for the new points. For each unlabeled data, a kNN query on the training set will be performed to estimate its class membership. This process can be considered as a kNN join of the testing set with the training set. The kNN operation can also be used to identify similar images. To do that, description features (points in a dataspace of dimension 128) are first extracted from images using a feature extractor technique. Then, the kNN operation is used to discover the points that are close, which should indicates similar images. Later in this paper, we consider this kind of data for the kNN computation. kNN join, together with other methods, can be applied to a large number of fields, such as multimedia [1], [2], social network [3], time series analysis [4], [5], bio-information and medical imagery [6], [7].

The basic idea to compute a kNN join is to perform a pairwise computation of distance for each element in R and each element in S. The difficulties mainly lie in the following two aspect: (1)Data Volume (2)Data Dimensionality. Suppose we are in a d dimension space, the computational complexity of this pairwise calculation is $O(d \times |R| \times |S|)$. Finding the k nearest neighbors in S for every r in R boils down to finding the smallest k distances, and leads to a minimum complexity of $|S| \times log\,|S|$. As the amount of data or their complexity (number of dimensions) increases, this approach becomes impractical. This is why a lot of work has been dedicated to reducing the in-memory computational complexity [8]–[12]. These works mainly focus on two points: (1) Using indexes to decrease the number of distances need to be calculated. However, these indexes can hardly be scaled on high dimension data. (2) Using projections to reduce the dimensionality of data. But the maintenance of the accuracy becomes another problem. Despite these efforts, there are still significant limitations to process kNN on a single machine when the amount of data increases. For large dataset, only distributed and parallel solutions prove to be powerful enough. MapReduce is a flexible and scalable parallel and distributed programming paradigm which is specially designed for data-intensive processing. It was first introduced by Google [13] and popularized with the Hadoop framework, an open source implementation.

The framework can be installed on commodity hardware and automatically distribute a MapReduce job over a set of machines. Writing an efficient kNN in MapReduce is also challenging for many reasons. First, classical algorithms as well as the index and projection strategies have to be redesigned to fit the MapReduce programming model and its share-nothing execution platform. Second, data partition and distribution strategies have to be carefully designed to limit communications and data transfer. Third, the load balancing problem which is new comparing with the single version should also be attached importance to. Then, not only the number of distance needed to be reduced, but also the number of MapReduce jobs and map/reduce tasks will bring impact. Finally, the parameter tuning remains always a key point to improve the performance.

The goal of this paper is to survey existing methods of kNN in MapReduce, and to compare their performance. It is a major extension of one of our previously published paper, [14], which provided only a simple theoretical analysis. Other surveys about kNN have been conducted, such as [15], [16], but they pursue a different goal. In [15], the authors only focus on centralized solutions to optimize kNN computation whereas we target distributed solutions. In [16], the survey is also oriented towards centralized techniques and is solely based on a theoretical performance analysis. Our approach comprehends both theoretical and practical performance analysis, obtained through extensive experiments. To the best of our knowledge, it is the first time such a comparison between existing kNN solutions on MapReduce has been performed. The breakthrough of this paper is that the solutions are experimentally compared in the same setting: same hardware and same dataset. Moreover, we present in this paper experimental settings and configurations that were not studied in the original papers. Overall, our contributions are:

- The decomposition of a distributed MapReduce kNN computation in different basic steps, introduced in Section 3.
- A theoretical comparison of existing techniques in Section 4, focusing on load balancing, accuracy and complexity aspects.
- An implementation of 5 published algorithms and an extensive set of experiments using both low and high dimension datasets (Section 5).
- An analysis which outlines the influence of various parameters on the performance of each algorithm.

The paper is concluded with a summary that indicates the typical dataset-solution coupling and provides precise guidelines to choose the best algorithm depending on the context.

2 CONTEXT

2.1 k Nearest Neighbors

A nearest neighbors query consists in finding at most k points in a data set S that are the closest to a considered point r, in a dimensional space d. More formally, given two data sets R and S in $\mathbf{R}^d$, and given r and s, two elements, with $r \in R$ and $s \in S$, we have:

Definition 1. Let $d(r, s)$ be the distance between r and s. The **kNN query** of r over S, noted $kNN(r, S)$ is the subset $\{s_i\} \subseteq S$ $(|\{s_i\}| = k)$, which is the k nearest neighbors of r in S, where $\forall\, s_i \in kNN(r, S), \forall\, s_j \in S{-}kNN(r, S)$, $d(r, s_i) \leq d(r, s_j)$.

This definition can be extended to a set of query points:

Definition 2. The **kNN join** of two datasets R and S, kNN($R \ltimes S$) is: kNN($R \ltimes S$)=$\{(r$,kNN(r,S)), $\forall\, r \in R\}$

Depending on the use case, it might not be necessary to find the exact solution of a kNN query, and that is why approximate kNN queries have been introduced. The idea is to have the k^{th} approximate neighbor not far from the k^{th} exact one, as shown in the following definition[1].

Definition 3. The $(1 + \epsilon)$**-approximate kNN query** for a query point r in a dataset S, $AkNN(r, S)$ is a set of approximate k nearest neighbors of r from S, if the k^{th} furthest result s^k satisfies $s^{k*} \leq s^k \leq (1 + \epsilon)s^{k*}$ $(\epsilon > 0)$ where s^{k*} is the exact k^{th} nearest neighbor of r in S.

And as with the exact kNN, this definition can be extended to an approximate kNN join AkNN($R \ltimes S$).

2.2 MapReduce

MapReduce [13] is a parallel programming model that aims at efficiently processing large datasets. This programming model is based on three concepts: (i) representing data as key-value pairs, (ii) defining a map function, and (iii) defining a reduce function. The map function takes key-value pairs as an input, and produces zero or more key-value pairs. Outputs with the same key are then gathered together (shuffled) so that key-{list of values} pairs are given to reducers. The reduce function processes all the values associated with a given key.

The most famous implementation of this model is the Hadoop framework [2] which provides a distributed platform for executing MapReduce jobs.

2.3 Existing kNN Computing Systems

The basic solution to compute kNN adopts a nested loop approach, which calculates the distance between every object r_i in R and s_j in S and sorts the results to find the k smallest. This approach is computational intensive, making it unpractical for large or intricate datasets. Two strategies have been proposed to work out this issue. The first one consists in reducing the number of distances to compute, by avoiding scanning the whole dataset. This strategy focuses on indexing the data through efficient data structures. For example, a one-dimension index structure, the B^+-Tree, is used in [17] to index distances; [18] adopts a multipage overlapping index structure R-Tree; [10] proposes to use a balanced and dynamic M-Tree to organize the dataset; [19] introduces a sphere-tree with a sphere-shaped minimum bound to reduce the number of areas to be searched; [20] presents a multidimensional quad-tree in order to be able to handle large amount of data; [12] develops a kd-tree which is a clipping partition method to separate the search space; and [21] introduces a loose coupling and shared nothing

[1] Erratum: this definition corrects the one given in the conference version of this journal.

[2] http://hadoop.apache.org/

distributed Inverted Grid Index structure for processing kNN query on MapReduce. However, reducing the searched dataset might not be sufficient. For data in large dimension space, computing the distance might be very costly. That is why a second strategy focuses on projecting the high-dimension dataset onto a low-dimension one, while maintaining the locality relationship between data. Representative efforts refer to LSH (Locality-Sensitive Hashing) [22] and Space Filling Curve [23].

But with the increasing amount of data, these methods still can not handle kNN computation on a single machine efficiently. Experiments in [24] suggest using GPUs to significantly improve the performance of distance computation, but this is still not applicable for large datasets that cannot reasonably be processed on a single machine. More recent papers have focused on providing efficient distributed implementations. Some of them use ad hoc protocols based on well-known distributed architectures [25], [26]. But most of them use the MapReduce model as it is naturally adapted for distributed computation, like in [27]–[29]. In this paper, we focus on the kNN computing systems based on MapReduce, because of its inherent scalability and the popularity of the Hadoop framework.

3 Workflow

We first introduce the reference algorithms that compute kNN over MapReduce. They are divided into two categories: (1) Exact solutions: The basic kNN method called hereafter **H-BkNNJ**; The block nested loop kNN named **H-BNLJ** [29]; A kNN based on Voronoi diagrams named **PGBJ** [28] and (2) Approximate solutions: A kNN based on z-value (a space filling curve method) named **H-zkNNJ** [29]; A kNN based on LSH, named **RankReduce** [27].

Although based on different methods, all of these solutions follow a common workflow which consists in three ordered steps: (i) data preprocessing, (ii) data partitioning and (iii) kNN computation. We analyze these three steps in the following sections.

3.1 Data Preprocessing

The idea of data preprocessing is to transform the original data to benefit from particular properties. This step is done before the partitioning of data to pursue two different goals: (1) either to reduce the dimension of data (2) or to select central points of data clusters.

To reduce the dimension, data from a high-dimensional space are mapped to a low-dimensional space by a linear or non-linear transformation. In this process, the challenge is to maintain the locality of the data in the low dimension space. In this paper, we focus on two methods to reduce data dimensionality. The first method is based on space filling curve. Paper [29] uses z-value as space-filling curve. The z-value of a data is a one dimensional value that is calculated by interleaving the binary representation of data coordinates from the most significant bit to the least significant bit. However, due to the loss of information during this process, this method can not fully guarantee integrity of the spatial location of data. In order to increase accuracy of this method, one can use several shifted copies of data and compute their z-values, although this increases the cost of computation and space. The second method to reduce data dimensionality is locality sensitive hashing (LSH) [22], [30]. This method maps the high-dimensional data into low-dimensional ones, with L families of M locality preserving hash functions $\mathcal{H}$ = { h(v) = $\lfloor \frac{a \cdot v+b}{W} \rfloor$ }, where a is a random vector, W is the size of the buckets into which transformed values will fall, and $b \in [0, W]$. And it makes sure that:

$$if\ d(x, y) \leq d_1, Pr_{\mathcal{H}}\left[h(x) = h(y)\right] \geq p_1 \quad (1)$$

$$if\ d(x, y) \geq d_2, Pr_{\mathcal{H}}\left[h(x) = h(y)\right] \leq p_2 \quad (2)$$

where d(x, y) is the distance between two points x and y, and $d_1 < d_2, p_1 > p_2$.

As a result, the closer two points x and y are, the higher the probability the hash values of these two points h(x) and h(y) in the hash family $\mathcal{H}$ (the set of hash functions used) are the same. The performance of LSH (how well it preserves locality) depends on the tuning of its parameters L, M, and W. The parameter L impacts the accuracy of the projection: increasing L increases the number of hash families that will be used, it thus increases the accuracy of the positional relationship by avoiding fallacies of a single projection, but in return, it also increases the processing time because it duplicates data. The parameter M impacts the probability that the adjacent points fall into the same bucket. The parameter W reflects the size of each bucket and thus, impacts the number of data in a bucket. All those three parameters are important for the accuracy of the result. Basically, the point of LSH for computing kNN is to have some collisions to find enough accurate neighbors. On this point, the reference RankReduce paper [27] does not highlight enough the cost of setting the right value for all parameters, and show only one specific setup that allow them to have an accuracy greater than 70%.

Another aspect of the preprocessing step can be to select central points of data clusters. Such points are called *pivots*. Paper [28] proposes 3 methods to select pivots. The *Random Selection* strategy generates a set of samples, then calculates the pairwise distance of the points in the sample, and the sample with the biggest summation of distances is chosen as set of pivots. It provides good results if the sample is large enough to maximize the chance of selecting points from different clusters. The *Furthest Selection* strategy randomly chooses the first pivot, and calculates the furthest point to this chosen pivot as the second pivot, and so on until having the desired number of pivots. This strategy ensures that the distance between each selected point is as large as possible, but it is more complex to process than the random selection method. Finally, the *K-Means Selection* applies the traditional k-means method on a data sample to update the centroid of a cluster as the new pivot each step, until the set of pivots stabilizes. With this strategy, the pivots are ensured to be in the middle of a cluster, but it is the most computational intensive strategy as it needs to converge towards the optimal solution. The quality of the selected pivots is crucial, for effectiveness of the partitioning step, as we will see in the experiments.

3.2 Data Partitioning and Selection

MapReduce is a shared-nothing platform, so in order to process data on MapReduce, we need to divide the dataset into independent pieces, called partitions. When computing a kNN, we need to divide R and S respectively. As in any MapReduce computation, the data partition strategy will strongly impact CPU, network communication and disk usages, which in turn will impact the overall processing time [31]. Besides, a good partition strategy could help to reduce the number of data replications, thereby reducing the number of distances needed to be calculated and sorted.

However, not all the algorithms apply a special data partition strategy. For example, H-BNLJ simply divides R into rows and S into lines, making each subset of R meeting with every subset of S. This ensures the distance between each object r_i in R and each object s_j in S will be calculated. This way of dividing datasets causes a lot of data replications. For example, in H-BNLJ, each piece of data is duplicated n times (n is the number of subsets of R and S), resulting in a total of n^2 tasks to calculate pairwise distances. This method wastes a lot of hardware resources, and ultimately leads a low efficiency.

The key to improve the performance is to preserve spatial locality of objects when decomposing data for tasks [32]. This means making a coarse clustering in order to produce a reduced set of neighbors that are candidates for the final result. Intuitively, the goal is to have a partitioning of data such that an element in a partition of R will have its nearest neighbors in only one partition of S. Two partitioning strategies that enable to separate the datasets into independent partitions, while preserving locality information, have been proposed. They are described in the two next sections.

3.2.1 Distance Based Partitioning Strategy

The distance based partitioning strategy we study in this paper is based on Voronoi diagram, a method to divide the space into disjoint cells. We can find other distance-based partitioning methods in the litterature, such as in [21], but we chose Voronoi diagram to represent distance-based partitioning method because it it can apply to data in any dimension. The main property of Voronoi diagram is that every point in a cell is closer to the pivot of this cell than to any other pivot. More formally, the definition of a Voronoi cell is as follow:

Definition 4. Given a set of disjoint pivots: P $=$ $\{p_1, p_2, ..., p_i, ..., p_n\}$, the Voronoi Cell of p_i $(0 < i \leq n)$ is: $\forall$ i $\neq$ j, $VC(p_i) = \{p \| d(p, p_i) \leq d(p, p_j)\}$.

Paper [28] gives a method to partition datasets R and S using Voronoi diagram. The partitioning principles are illustrated in Figure 1. After having identified the pivots p_i in R (c.f. Section 3.1), the distances between elements of each dataset and the pivots are computed. The elements are then put in the cell of the closest pivot, giving a partitioning of R (resp. S) into P_i^R (resp. P_i^S). For each cell, the upper bound $U(P_i^R)$ (resp. the lower bound $L(P_i^R)$) is computed as a sphere determined by the furthest (resp. nearest) point in P_i^R from the pivot p_i. The boundaries and other statistics are used to find candidate data from S in the neighboring cells. These data are then replicated in cell P_i^S. For example, in Figure 1, the element s of P_j^S falls inside $U(P_i^R)$ and is thus copied to S_i as a potential candidate for the kNN of r.

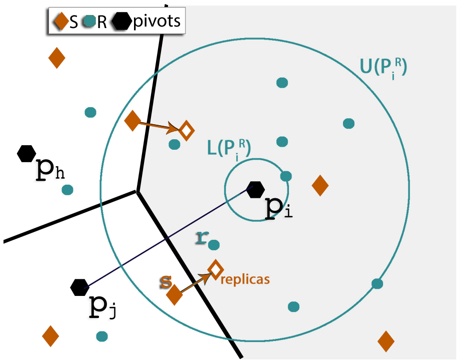


Fig. 1: Voronoi : cells partitioning and replication for cell P_i

The main issue with this method is that it requires computing the distance from all elements to the pivots. Also, the distribution of the input data might not be known in advance. Hence, pivots have to be recomputed if data change. More importantly, there is no guarantee that all cells have an equal number of elements because of potential data skew. This can have a negative impact on the overall performance because of load balancing issues. To alleviate this issue, the authors propose two grouping strategies, which will be discussed in Section 4.1.

3.2.2 Size Based Partitioning Strategy

Another type of partitioning strategy aims at dividing data into equal size partitions. Paper [29] proposes a partitioning strategy based on z-value described in the previous section.

In order to have a similar number of elements in all n partitions, the authors first sample the dataset and compute the n quantiles. These quantiles are an unbiased estimation of the bounds for each partition. Figure 2 shows an example for this method. In this example data are only shifted once. Then, data are projected using z-value method, and the interpolating "Z" in the figure indicates the neighborhood after projection. Data is projected into a one dimension space, represented by Z_i^R and Z_i^S in the figure. Z_i^R is divided into partitions using the sampling estimation explained above. For a given partition R_i, its corresponding S_i is defined in Z_i^S by copying the nearest k preceding and the nearest k succeeding points to the boundaries of S_i. In Figure 2, four points of S_i are copied in partition 2, because they are considered as candidates for the query points in R_i^2.

This method is likely to produce a substantially equivalent number of objects in each partition, thus naturally achieving load balancing. However, the quality of the result depends solely on the quality of the z-curve, which might be an issue for high dimension data.

Another similar size based partitioning method uses Locality Sensitive Hashing to first project data into low dimension space as illustrated in Figure 3. In this example, data is hashed twice using two hash families $a1$ and $a2$. Each hashed data is then projected in the corresponding bucket. The principle of this method is to result in collisions for data that is spatially close. So the data initially close in the high dimension space is hashed to the same bucket with a high probability, provided that the bucket size (parameter W in LSH) is large enough to receive at least one copy of close data.

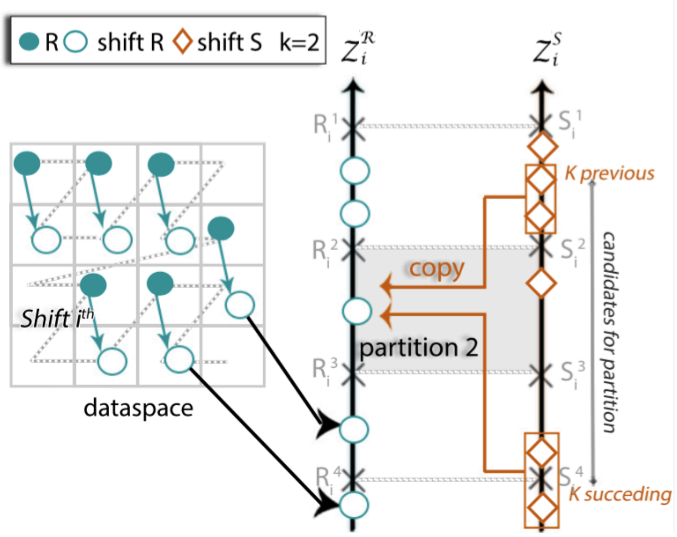


Fig. 2: z-value : partition

The strategy of partitioning directly impacts on the number of tasks and the amount of computation. Distance based methods aim at dividing the space into cells that are driven by distance rules. Size based methods create equal size zones in which the points are ordered. Regarding the implementation, [28] uses a MapReduce job to perform the partitioning. In [29], both data preprocessing and data partitioning are completed in a single MapReduce job.

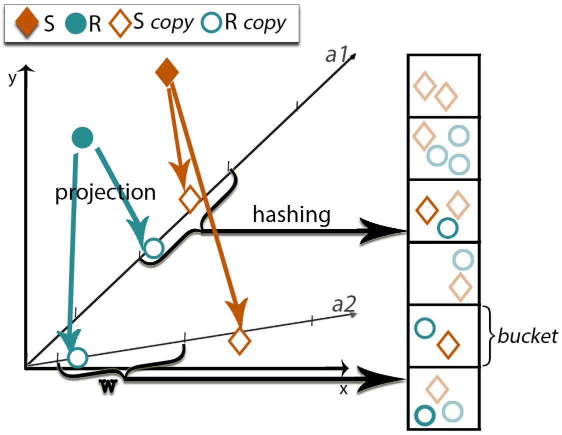


Fig. 3: LSH : bucket

3.3 Computation

The main principle to compute a kNN, is to (i) calculate the distance between r_i and s_j for all i, j, and (ii) sort these distances in ascending order to pick the first k results. The number of MapReduce jobs for computing and sorting has a significant impact on the global performance of the kNN computation, given the complexity of MapReduce task and the amount of data to exchange between them. The preprocessing and partitioning steps impact on the number of MapReduce tasks that are further needed for the core computation. In this section, we review the different strategies used to finally compute and sort distances efficiently using MapReduce. These different strategies can be divided into two categories, depending on the number of jobs they require. Those categories can themselves be divided into two subcategories: the ones that do not preprocess and partition data before computation and the ones that implement the preprocessing and partitioning steps.

3.3.1 One MapReduce Job

Without preprocessing and partitioning strategies.
The naive solution (**H-BkNNJ**) only uses one MapReduce job to calculate and sort the distances, and only the Map Phase is done in parallel. The Map tasks will cut datasets into splits, and label each split with its original dataset (R or S). The Reduce task then takes one object r_i and one object s_j to form a key-value pair $< r_i, s_j >$, and calculate the distance between them, then for each key r_i sort the distances with every object in S, leading the number of distances need to be sorted to $|S|$. Since only the Map phase is in parallel, and only one Reduce task is used for calculating and sorting, when the datasets becomes large, this method will quickly exceed the processing capacity of the computer. Therefore, it is only suitable for small datasets.

With preprocessing and partitioning strategies.
PGBJ [28] uses a preprocessing step to select the pivot of each partition and a distance based partitioning strategy to ensure that each subset R_i only needs one corresponding subset S_i to form a partition where the kNN of all $r_i \in R_i$ can be found. Therefore, in the computation step, Map tasks find the corresponding S_i for each R_i according to the information provided by the partitioning step. Reduce tasks then perform the kNN join inside each partition of $< R_i, S_i >$.

Overall, the main limitation of these two approaches is that the number of values to be sorted in the Reduce task can be extremely large, up to $|S|$, if the preprocessing and partitioning steps have not significantly reduced the set of searched points.

This aspect can limit the applicability of such approaches in practice.

3.3.2 Two Consecutive MapReduce Jobs

To overcome the previously described limitation, multiple successive MapReduce jobs are required. The idea is to have the first job output the local top k for each pair (R_i, S_j). Then, the second job is used to merge all the top k values for a given r_i and to merge and sort all local top k values (instead of all values) producing the final global top k.

Without preprocessing and partitioning strategies.
H-BNLJ does not have any special preprocessing or partitioning strategy. The Map Phase of the first job distributes R into n rows and S into n columns. The n^2 Reduce tasks output the local kNN for each object r_i in the form of $(r_{id}, s_{id}, d(r, s))$.

Since each r_{id} has been replicated n times, the Map Phase of the second MapReduce job will pull every candidate of r_i from the n pieces of R, and form $(r_{id}), list(s_{id}, d(r, s))$. Then each Reduce task will sort $list(s_{id}, d(r, s))$ in ascending order of $d(r, s)$ for each r_i, and finally, give the top k results.

Moreover, in order to avoid the scan of the whole dataset of each block, some index structures like R-Tree [29] or Hilbert R-Tree [33] can be used to index the local S blocks.

With preprocessing and partitioning strategies.

In **H-zkNNJ** [29] the authors propose to define the bounds of the partitions of R and then to determine from this the corresponding S_i in a preprocessing job. So here, the preprocessing and partitioning steps are completely integrated in MapReduce. Then, a second MapReduce job takes the partitions R_i and S_i previously determined, and computes for all r_i the candidate neighbor set, which represents the points that could be in the final kNN[3]. To get this candidate neighbor set, the closest k points are taken from either side of the considered point (the partition is in dimension 1), which leads to exactly $2k$ candidate points. The third MapReduce round determines the exact result for each r_i from the candidate neighbor set. So in total, this solution uses three MapReduce jobs, and among them, the last two are actually devoted to the kNN core computation. As the number of points that are in the candidate neighbor set is small (thanks to the drastic partitioning, itself due to a drastic preprocessing), the cost of computation and communication is extremely reduced.

In **RankReduce** [27][4], the authors first preprocess data to reduce the dimentionality and partition data into buckets using LSH. In our implementation, like in **H-zkNNJ**, one MapReduce job is used to calculate the local kNN for each r_i, and a second one is used to find the global ones.

3.4 Summary

So far, we have studied the different ways to go through a kNN computation from a workflow point of view with three main steps. The first step focuses on data preprocessing, either for selecting dominating points or for projecting data from high dimension to low dimension. The second step aims at partitioning and organizing data such that the following kNN core computation step is lighten. This last step can use one or two MapReduce jobs depending on the number of distances we want to calculate and sort. Figure 4 summarizes the workflow we have gone through in this section and the techniques associated with each step.

4 Theoretical Analysis

4.1 Load Balance

In a MapReduce job, the Map tasks or the Reduce tasks will be processed in parallel, so the overall computation time of each phase depends on the completion time of the longest task. Therefore, in order to obtain the best performance, it is important that each task performs substantially the same amount of computation. When considering load balancing in this section, we mainly want to have the same time complexity in each task. Ideally, we want to calculate roughly the same number of distance between r_i and s_j in each task.

For **H-BkNNJ**, there is no load balancing problem. Because in this basic method, only the Map Phase is treated in parallel. In Hadoop each task will process 64M data by default.

H-BNLJ cuts both the dataset R and the dataset S into p equal-size pieces, then those pieces are combined pairwise to form a partition of $< R_i, S_j >$. Each task will process one block of data so we need to ensure that the size of the data block handled by each task is roughly the same. However, **H-BNLJ** uses a random partitioning method which can not exactly divide the data into equal-size blocks.

PGBJ uses Voronoi diagram to cut the data space of R into cells, where each cell is represented by its pivot. Then the data are assigned to the cell whose pivot is the nearest from it. For each R cell, we need to find the corresponding pieces of data in S. Sometimes, the data in S may be potentially needed by more than one R cells, which will lead to the duplication of some elements of S. Thus the number of distances to be calculated in each task, i.e. the relative time complexity of each task is:

$$\mathcal{O}\left(Task\right) = \left|P_i^R\right| \times \left(\left|P_i^S\right| + \left|RepS_c\right|\right)$$

where $\left|P_i^R\right|$ and $\left|P_i^S\right|$ represents the number of elements in cell P_i^R or P_i^S respectively, and $|RepS_c|$ the number of replicated elements for the cell. Therefore, to ensure load balancing, we need to ensure that $\mathcal{O}\left(Task\right)$ is roughly the same for each task. **PGBJ** introduces two methods to group the cells together to form a bigger cell which is the input of a task. On one hand, the *geo grouping* method supposes that close cells have a higher probability to replicate the same data. On the other hand, the *greedy grouping* method estimates the cells whose data are more likely to be replicated. This approximation gives an upper bound on the complexity of the computation for a particular cell, which enables grouping of the cells that have the most replicated data in common. This leads to a minimization of replication and leads to groups that generate the same workload.

And the **H-zkNNJ** method assumes:

$$\begin{gathered}\forall \text{ i} \neq \text{j}, \\ \text{if } |R_i| = |R_j| \text{ or } |S_i| = |S_j|, \\ \text{then } |R_i| \times |S_i| \approx |R_j| \times |S_j|\end{gathered}$$

That is to say, if the number of objects in each partition of R is equivalent, then the sum of the number of k nearest neighbors of all objects in each partition can be considered approximately equivalent, and vice versa. So an efficient partitioning should try to enforce either (i) $|R_i| = |R_j|$ or (ii) $|S_i| = |S_j|$. In paper [29], the authors give a short proof which shows that the worst-case computational complexity for (i) is equal to:

$$\mathcal{O}\left(|R_i| \times log\,|S_i|\right) = \mathcal{O}\left(\frac{|R|}{n} \times log\,|S|\right) \tag{3}$$

and for choice (ii), the worst-case complexity is equal to:

$$\mathcal{O}\left(|R_i| \times log\,|S_i|\right) = \mathcal{O}\left(|R| \times log\frac{|S|}{n}\right) \tag{4}$$

where n is the number of partitions. Since $n \ll |S|$, the optimal partitioning is achieved when $|R_i| = |R_j|$.

In **RankReduce**, a custom partitioner is used to load balance tasks between reducers. Let $W_h = \mid R_h \mid \times \mid S_h \mid$ be the weight of bucket h. A bin packing algorithm is used such that each reducer ends up with approximately the same amount of work. More precisely, let $\mathcal{O}\left(R_i\right) = \sum_h W_h$ the work done by reducer Ri, then this methods guarantees that

$$\forall i \neq j, \mathcal{O}\left(R_i\right) \approx \mathcal{O}\left(R_j\right) \tag{5}$$

[3] Note that the notion of candidate points is different from local top k points.

[4] Although RankReduce only computes kNN for a single query, it is directly expandable to a full kNN join.

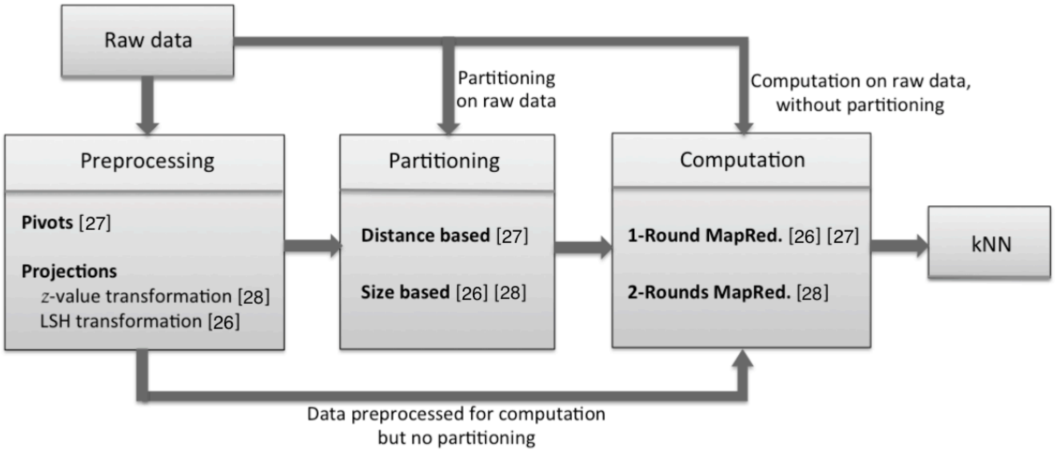


Fig. 4: Usual workflow of a kNN computation using MapReduce

Because the weight of a bucket is only an approximation of the computing time, this method can only give an approximate load balance. Having a large number of buckets compared to the number of reducers significantly improves the load balancing.

4.2 Accuracy

Usually, the lack of accuracy is the direct consequence of techniques to reduce the dimensionality with techniques such as z-values and LSH. In [29] (**H-zkNNJ**), the authors show that when the dimension of the data increases, the quality of the results tends to decrease. This can be counterbalanced by increasing the number of random shifts applied to the data, thereby increasing the size of the resulting dataset. Their experiments show that three shifts of the initial dataset (in dimension 15) are sufficient to achieve a good approximation (less than 10% of errors measured), while controlling the computation time. Furthermore, paper [34] shows a detailed theoretical analyses showing that, for any fixed dimension, by using only $\mathcal{O}(1)$ random shifts of data, the z-value method returns a constant factor approximation in terms of the radius of the k nearest neighbor ball.

For LSH, the accuracy is defined by the probability that the method will return the real nearest neighbor. Suppose that the points within a distance $d = |p - q|$ are considered as close points. The probability [35] that these two points end up in the same bucket is:

$$p(d) = Pr_{\mathcal{H}}\left[h(p) = h(q)\right] = \int_0^W \frac{1}{d} f_s(\frac{x}{d})(1 - \frac{x}{W})dx \tag{6}$$

where W is the size of the bucket and f_s is the probability density function of the hash function $\mathcal{H}$. From this equation we can see that for a given bucket size W, this probability decreases as the distance d increases. Another way to improve the accuracy of LSH is to increase the number of hashing families used. The use of LSH in **RankReduce** has an interesting consequence on the number of results. Depending on the parameters, the number of elements in a bucket might be smaller than k. Overall, unlike z-value, the performance of LSH depends a lot on parameter tuning.

4.3 Global Complexity

Carefully balancing the number of jobs, tasks, computation and communication is an important part of designing an efficient distributed algorithm. All the kNN algorithms studied in this survey have different characteristics. We will now describe them and outline how they can impact the execution time.

(1) **The number of MapReduce jobs:** Starting a job (whether in Hadoop [36] or any other platform) requires some initialization steps such as allocating resources and copying data. Those steps can be very time consuming.

(2) **The number of Map tasks and Reduce tasks used to calculate kNN$(R_i \ltimes S)$:** The larger this number is, the more information is exchanged through the network during the shuffle phase. Moreover, scheduling a task also incurs an overhead. But the smaller this number is, the more computation is done by one machine.

(3) **The number of final candidates for each object r_i:** We have seen that advanced algorithms use preprocessing and partitioning techniques to reduce this number as much as possible. The goal is to reduce the amount of data transmitted and the computational cost.

Together these three points impact two main overheads that affect the performance:

- Communication overhead, which can be considered as the amount of data transmitted over the network during the shuffle phases.
- Computation overhead, which is mainly composed of two parts: 1). computing the distances, 2). finding the k smallest distances. It is also impacted by the complexity (dimension) of the data.

Suppose the dataset is d dimensional, the overhead for computing the distance is roughly the same for every r_i and s_j for each method. The difference comes from the number of distances to sort for each element r_i to get the top k nearest neighbors. Suppose that the dataset R is divided into n splits. Here n represents the number of partitions of R and S for **H-BNLJ** and **H-zkNNJ**, the number of cells after using the grouping strategy for **PGBJ** and the number of buckets for **RankReduce**. Assuming there is a good load balance for each method, the number of elements in one split R_i can be considered as $\frac{|R|}{n}$. Finding the k closest neighbors efficiently for a given r_i can be done using a priority queue, which less costly than sorting all candidates.

Since all these algorithms uses different strategies, their steps cannot be directly compared. Nonetheless, to provide a theoretical insight, we will now compare their complexity for the last phase, which is common to all of them.

The basic method **H-BkNNJ** only uses one MapReduce job, and requires only one Reduce task to compute and sort the distances. The communication overhead is $\mathcal{O}(|R| + |S|)$. The number of final candidates for one r_i is $|S|$. The complexity of finding the k smallest distances for r_i is $\mathcal{O}(|S| \cdot log\,(k))$. Hence, the total cost for one task is $\mathcal{O}(|R| \cdot |S| \cdot \log\,(k))$. Since R and S are usually large datasets, this method quickly becomes impracticable.

To overcome this limitation, **H-BNLJ** [29] uses two MapReduce jobs, with n^2 tasks to compute the distances. Using a second job significantly reduces the number of final candidates to nk. The total communication overhead is $\mathcal{O}(n\,|R| + n\,|S| + kn\,|R|)$. The complexity of finding the k elements for each r_i is reduced to $(n \cdot k) \cdot log\,(k)$. Since each task has $\frac{|R|}{n}$ elements, the total sort overhead for one task is $\mathcal{O}(|R| \cdot k \cdot log(k))$.

PGBJ [28] performs a preprocessing phase followed by two MapReduce jobs. This method also only uses n Map tasks to compute the distances and the number of final candidates falls to $|S_i|$. Since this method uses a distance based partitioning method, the size of $|S_i|$ varies, depending on the number of cells required to perform the computation and the number of replications ($|RepS_c|$, see Section 4.1) required by each cell. As such, the computational complexity cannot be expressed easily.Overall, finding the k elements is reduced to $\mathcal{O}(|S_i| \cdot log\,|k|)$ for each r_i, and $\mathcal{O}(\frac{|R|}{n} \cdot |S_i| \cdot log\,|S_i|)$ in total per task. The communication overhead is $\mathcal{O}(|R| + |S| + |RepS_c| \cdot n)$. In the original paper the authors gives a formula to compute $|RepS_c| \cdot n$, which is the total number of replications for the whole dataset S.

In **RankReduce** [27], the initial dataset is projected by L hash families into buckets. After finding the local candidates in the second job, the third job combines the local results to find the global k nearest neighbor. For each r_i, the number of final candidates is $L \cdot k$. Finding the k elements takes $\mathcal{O}(L \cdot k \cdot log(k))$ per r_i, and $\mathcal{O}(|R_i| \cdot L \cdot k \cdot log(k))$ per task. The total communication cost becomes $\mathcal{O}(|R| + |S| + k \cdot |R|)$.

H-zkNNJ [29] also begins by a preprocessing phase and uses in total three MapReduce jobs in exchange for requiring only n Map tasks. For a given r_i, these tasks process elements from the candidate neighbor set $C\,(r_i)$. By construction, $C\,(r_i)$ only contains $\alpha \cdot k$ neighbors, where α is the number of shifts of the original dataset. The complexity is now reduced to $\mathcal{O}((\alpha \cdot k) \cdot log\,(k))$ for one r_i, and $\mathcal{O}(\frac{|R|}{n} \cdot (\alpha \cdot k) \cdot log\,(k))$ in total per task. The communication overhead is $\mathcal{O}(\frac{1}{\varepsilon^2} + |S| + k \cdot |R|)$, with $\varepsilon \in (0, 1)$, a parameter of the sampling process.

From the above analysis we can infer the following. **H-BkNNJ** only uses one task, but this task needs to calculate the entire data set. **H-BNLJ** uses n^2 tasks to greatly reduce the amount of data processed by each task. However this also increases the amount of data to be exchanged among the nodes. This should prove to be a major bottleneck. **PGBJ**, **RankReduce** and **H-zkNNJ** all use three jobs which reduces the number of tasks to n, and thus reduces the communication overhead.

Although the computational complexity of each task depends on various parameters of the preprocessing phases, it is possible to outline a partial conclusion from this analysis. There are basically three performance brackets. First, the least efficient should be **H-BkNNJ**, followed by **H-BNLJ**. **PGBJ**, **RankReduce** and **H-zkNNJ** are theoretically the most efficient. Among them, **PGBJ** has the largest number of final candidates. For **RankReduce** and **H-zkNNJ** , the number of final candidates is of the same order of magnitude. The main difference lies in the communication complexity, more precisely in $\frac{1}{\varepsilon^2}$ compared to $|R|$. As the dataset size increases, we will have eventually $|R| \gg \frac{1}{\varepsilon^2}$. Hence, **H-zkNNJ** seems to be the theoretically most efficient for large query sets.

4.4 Wrap up

Although the workflow for computing kNN on MapReduce is the same for all existing solutions, the guarantees offered by each of them vary a lot. As load balancing is a key point to reduce completion time, one should carefully choose the partitioning method to achieve this goal. Also, the accuracy of the computing system is crucial: are exact results really needed? If not, then one might trade accuracy for efficiency, by using data transformation techniques before the actual computation. Complexity of the global system should also be taken into account for particular needs, although it is often related to the accuracy: an exact system is usually more complex than an approximate one. Table 1 shows a summary of the systems we have examined and their main characteristics.

Due to the multiple parameters and very different steps for each algorithm, we had to limit our complexity analysis to common operations. Moreover, for some of them, the complexity depends on parameters set by a user or some properties of the dataset. Therefore, the total processing time might be different, in practice, than the one predicted by the theoretical analysis. That is why it is important to have a thorough experimental study.

5 Evaluation

In order to compare theoretical performance and performance in practice, we performed an extensive experimental evaluation of the algorithms described in the previous sections.

The experiments were run on two clusters of Grid'5000[5], one with Opteron 2218 processors and 8GB of memory, the other with Xeon E5520 processors and 32GB of memory, using Hadoop 1.3, 1Gb/s Ethernet and SATA hard drives. We follow the default configuration of Hadoop: (1) the number of replications for each split of data is set to 3; (2) the number of slot of each node is 1, so only one map/reduce task is processed on the node at one time.

We evaluate the five approaches presented before.

[5] www.grid5000.fr

Methods	Preprocessing	Partitioning	Accuracy	Complexity			
				Jobs	Tasks	Final Candidate (per r_i)	Communication
H-BkNNJ (Basic Method)	None	None	Exact	1	1	$\|S\|$	$\mathcal{O}(\|R\| + \|S\|)$
H-BNLJ [29] (Zhang et al.)	None	None	Exact	2	n^2	nk	$\mathcal{O}(n\|R\| + n\|S\| + kn\|R\|)$
PGBJ [28] (Lu et al.)	Pivots Selection	Distance Based	Exact	3	n	$\|S_i\|$	$\mathcal{O}(\|R\| + \|S\| + \|RepS_c\| \cdot n)$
RankReduce [27] (Stupar et al.)	LSH	Size Based	Approximate	3	n	$L \cdot k$	$\mathcal{O}(\|R\| + \|S\| + k \cdot \|R\|)$
H-zkNNJ [29] (Zhang et al.)	Z-Value	Size Based	Approximate	3	n	$\alpha \cdot k$	$\mathcal{O}(\frac{1}{\varepsilon^2} + \|S\| + k \cdot \|R\|)$

TABLE 1: Summary table of kNN computing systems with MapReduce

For **H-zkNNJ** and **H-BNLJ**, we took the source code provided by the authors as a starting point[6] and added some modifications to reduce the size of intermediate files. The others were implemented from scratch using the description provided in their respective papers.

When implementing **RankReduce**, we added a reduce phase to the first MapReduce job to compute some statistics information for each bucket. These information is used for achieving good load balance. Moreover, in the original version, the authors only use one hash function. To improve the precision, we choose to use multiple families and hash functions depending on the dataset. Finally, our version of **RankReduce** uses three MapReduce jobs instead of two.

Most of the experiments were ran using two different datasets:

- **OpenStreetMap:** we call it the *Geographic - or Geo - dataset*. The Geo dataset involves geographic XML data in two dimensions[7]. This is a real dataset containing the location and description of objects. The data is organized by region. We extract $256 * 10^5$ records from the region of France.
- **Catech 101:** we call it the *Speeded Up Robust Features - or SURF - dataset*. It is a public set of images[8], which contains 101 categories of pictures of objects, and 40 to 800 images per category. SURF [37] is a detector and a descriptor for points of interest in images, which produces image data in 128 dimensions. We extract 32 images per category, each image has between 1000 and 2000 descriptors.

In order to learn the impact of dimension and dataset, we use 5 additional datasets: **El Nino:** in 9 dimensions; **HIGGS:** in 28 dimensions; **TWITTER:** in 77 dimensions; **BlogFeedBack:** in 281 dimensions; and **Axial Axis:** in 386 dimensions. These data sets are all downloaded from UCI Machine Learning Repository[9].

We use a self-join for all our experiments, which means we use two datasets of equal sizes for R and S ($|R| = |S|$). We also vary the number of records of each dataset from $0.125 * 10^5$ to $256 * 10^5$. For all experiments, we have set $k = 20$ unless specified otherwise.

We study the methods from the following aspects:

- The impact of data size
- The impact of k
- The impact of dimension and dataset

And we record the following information: the processing time, the disk space required, the recall and precision, and the communication overhead.

To assess the quality of the approximate algorithms, we compute two commonly used metrics and use the results of the exact algorithm **PGBJ** as a reference. First, we define the recall as $recall = \frac{|A(v) \bigcap I(v)|}{|I(v)|}$, where $I(v)$ are the exact kNN of v and $A(v)$ the kNN found by the approximate methods. Intuitively, the recall measures the ability of an algorithm to find the correct kNNs. Another metric, the precision is defined by $precision = \frac{|A(v) \bigcap I(v)|}{|A(v)|}$. It measures the fraction of correct kNN in the final result set. By definition, the following properties holds: (1) $recall \leq precision$ because all the tested algorithms return up to k elements. (2) if an approximate algorithms outputs k elements, then $recall = precision$.

Each algorithm produces intermediate data so we compute a metric called *Space requirement* based on the size of intermediate data ($Size_{intermediate}$), the size of the result ($Size_{final}$) and the size of the correct kNN ($Size_{correct}$). We thus have $space = \frac{Size_{final} + Size_{intermediate}}{Size_{correct}}$.

We start by evaluating the most efficient number of machines to use (hereafter called *nodes*) in terms of resources and computing time. For that, we measure the computing time of all algorithm for three different data input size of the geographic dataset. The result can be seen on Figure 5. As expected, the computing time is strongly related to the number of nodes. Adding more nodes increases parallelism, reducing the overall computing time. There is however a significant slow down after using more than 15 machines. Based on those results, and considering the fact that we later use larger datasets, we conducted all subsequent experiments using at most 20 nodes.

5.1 Geographic dataset

For all experiments in this section, we used the parameters described in Table 2. Details regarding each parameter can

[6] http://ww2.cs.fsu.edu/~czhang/knnjedbt/
[7] Taken from: http://www.geofabrik.de/data/download.html
[8] Taken from: www.vision.caltech.edu/Image_Datasets/Caltech101
[9] Taken from: https://archive.ics.uci.edu/ml/

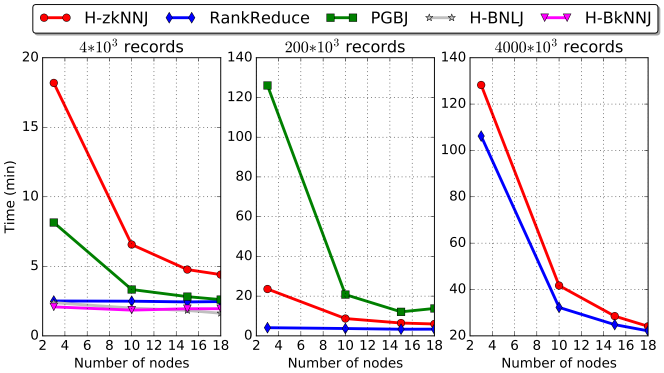


Fig. 5: Impact of the number of nodes on computing time

be found in sections 3.1 and 3.2. For RankReduce, the value of W was adapted to get the best performance from each dataset. For datasets up to $16 * 10^5$ records, $W = 32 * 10^5$, up to $25*10^5$ records, $W = 25*10^5$ and finally, $W = 15*10^5$ for the rest of the experiments.

Algorithm	Partitioning	Reducers	Configuration
H-BNLJ	10 partitions	100 reducers	
PGBJ	3000 pivots	25 reducers	k-means + greedy

RankReduce	$W = \begin{cases} 32 * 10^5 \\ 25 * 10^5 \\ 15 * 10^5 \end{cases}$	25 reducers	L = 2
			M = 7

H-zkNNJ	10 partitions	30 reducers	3 shifts, p=10

TABLE 2: Algorithm parameters for geographic dataset

5.1.1 Impact of input data size

Our first set of experiments measures the impact of the data size on execution time, disk space and recall. Figure 6a shows the global computing time of all algorithms, varying the number of records from $0.125 * 10^5$ to $256 * 10^5$. The global computing time increases more or less exponentially for all algorithms, but only **H-zkNNJ** and **RankReduce** can process medium to large datasets. For small datasets, **PGBJ** can compute an exact solution as fast as the other algorithms.

Figure 6b shows the space requirement of each algorithm as a function of the final output size. To reduce the footprint of each run, intermediate data are compressed. For example, for **H-BNLJ**, the size of intermediate data is 2.6 times bigger than the size of output data. Overall, the algorithms with the lowest space requirements are **RankReduce** and **PGBJ**.

Figure 6c shows the recall and precision of the two approximate algorithms, **H-zkNNJ** and **RankReduce**. Since **H-zkNNJ** always return k elements, its precision and recall are identical. As the number of records increases, its recall decreases, while still being high, because of the space filling curves used in the preprocessing phase. On the other hand, the recall of **RankReduce** is always lower than its precision because it outputs less than k elements. It benefits from larger datasets because more data end up in the same bucket, increasing the number of candidates. Overall, the quality of **RankReduce** was found to be better than **H-zkNNJ** on the Geo dataset.

5.1.2 Impact of k

Changing the value of k can have a significant impact on the performance of some of the kNN algorithms. We experimented on a dataset of $2 * 10^5$ records (only $5 * 10^4$ for **H-BNLJ** for performance reasons) with values for k varying from 2 to 512. Results are shown in Figure 7 using a logarithmic scale on the x-axis.

First, we observe a global increase in computing time (Figure 7a) which matches the complexity analysis performed earlier. As k increases, the performance of **H-zkNNJ**, compared to the other advanced algorithms, decreases. This is due to the necessary replication of the z-values of S throughout the partitions to find enough candidates: the core computation is thus much more complex.

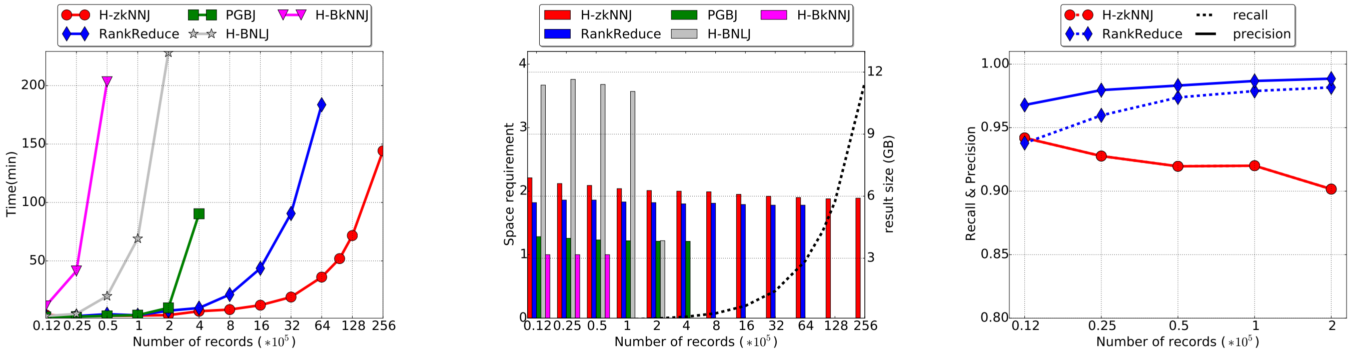
Second, the algorithms can also be distinguished considering their disk usage, visible on Figure 7b. The global tendency is that the ratio of intermediate data size over the final data size decreases. This means that for each algorithm the final data size grows faster than the intermediate data size. As a consequence, there is no particular algorithm that suffers from such a bottleneck at this point. **PGBJ** is the most efficient from this aspect. Its replication of data occurs independently of the number of selected neighbors. Thus, increasing k has a small impact on this algorithm, both in computing time and space requirements. On this figure, an interesting observation can also be made for **H-zkNNJ**. For $k = 2$, it has by far the largest disk usage but becomes similar to the others for larger values. This is because **H-zkNNJ** creates a lot of intermediate data (copies of the initial dataset, vectors for the space filling curve, sampling...) irrespective of the value of k. As k increases, so does the output size, mitigating the impact of these intermediate data.

Surprisingly, changing k has a different impact on the recall of the approximate kNN methods, as can be seen on Figure 7c. For **RankReduce**, increasing k has a negative impact on the recall which sharply decreases when $k \geq 64$. This is because the window parameter (W) of LSH was set at the beginning of the experiments to achieve the best performance for this particular dataset. However, it was not modified for various of k. Thus it became less optimal as k increased. This shows there is a link between global parameters such as k and parameters of the LSH process. When using **H-zkNNJ**, increasing k improves the precision: the probability to have incorrect points is reduced as there are more candidates in a single partition.

5.1.3 Communication Overhead

Our last set of experiments looks at inter-node communication by measuring the amount of data transmitted during the shuffle phase (Figure 8). The goal is to compare these measurements with the theoretical analysis in Section 4.3,

Impact of data size. For Geo dataset (Figure 8a), **H-BNLJ** has indeed a lot of communication. For a dataset of $1 * 10^5$ records, the shuffle phase transmits almost 4 times the original size. Both **RankReduce** and **H-zkNNJ** have a constant factor of 1 because of the duplication of the original dataset to improve the recall. The most efficient algorithm is **PGBJ** for two reasons. First it does not duplicate the original dataset and second, it relies on various grouping strategies to minimize replication.

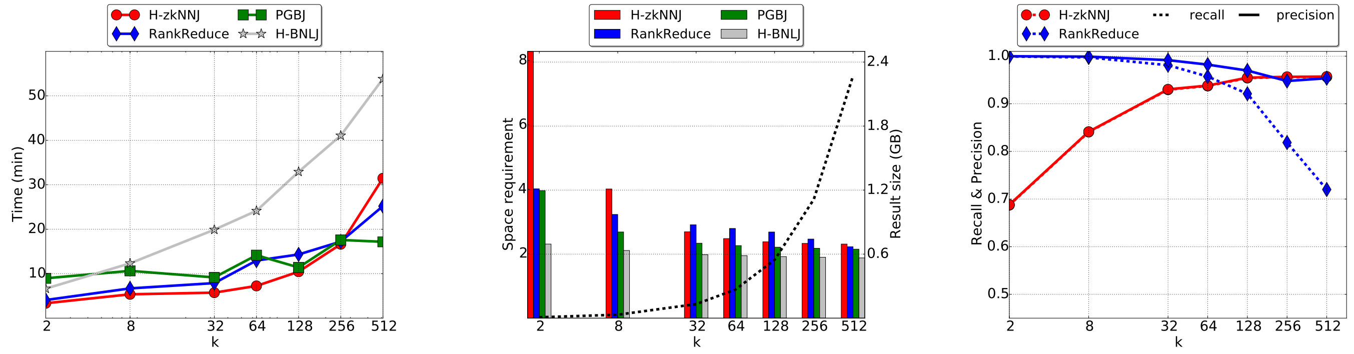


(a) Time

(b) Result size and Disk Usage

(c) Recall and Precision

Fig. 6: Geo dataset impact of the data set size

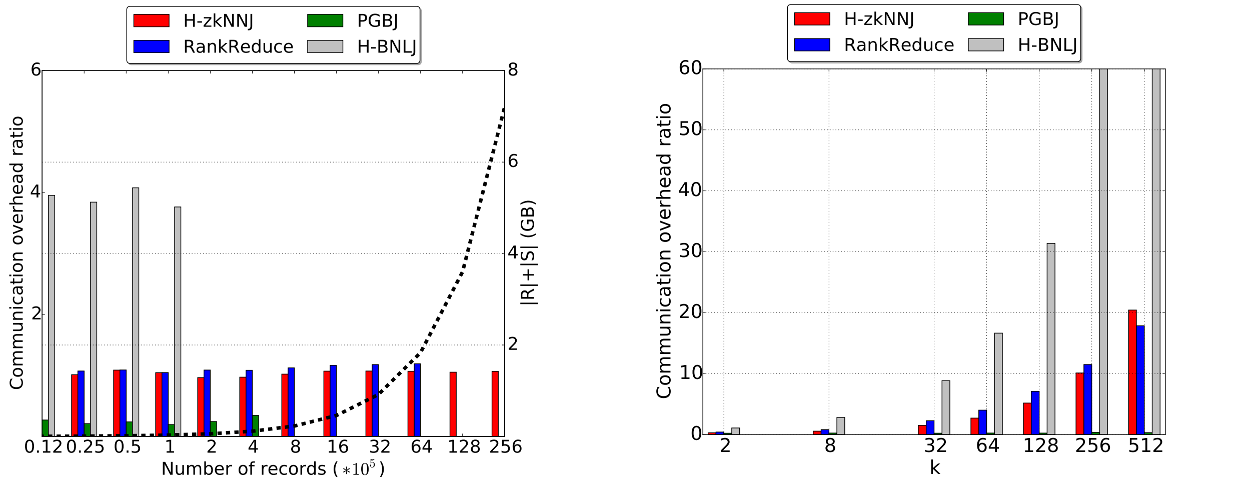


(a) Time

(b) Result size and Disk Usage

(c) Recall and Precision

Fig. 7: Geo dataset with 200k records (50k for H-BNLJ), impact of K



(a) Impact of the data set size

(b) Impact of K with $2 * 10^5$ records ($0.5 * 10^5$ for H-BNLJ)

Fig. 8: Geo dataset, communication overhead

Impact of k. We have performed another set of experiments, with a fixed dataset of $2 * 10^5$ records (only $0.5 * 10^5$ for **H-BNLJ**). The results can be seen in Figure 8b. For different values of k, we have a similar hierarchy than with the data size. For **RankReduce** and **H-zkNNJ**, the shuffle increases linearly because the number of candidates in the second phase depends on k. Moreover **H-zkNNJ** also replicates k previous and succeeding elements in the first phase, and because of that, its overhead becomes significant for large k. Finally in **PGBJ**, k has no impact on the shuffle phase.

5.2 Image Feature Descriptors (SURF) dataset

We now investigate whether the dimension of input data has an impact on the kNN algorithms using the SURF dataset. We used the Euclidian distance between descriptors to measure image similarity. For all experiments in this section, the parameters mentioned in Table 3 are used.

Algorithm	Partitioning	Reducers	Configuration
H-BNLJ	10 partitions	100 reducers	
PGBJ	3000 pivots	25 reducers	k-means + geo

RankReduce	W = 10^7	25 reducers	L = 5
			M = 7

H-zkNNJ	6 partitions	30 reducers	5 shifts

TABLE 3: Algorithm parameters for SURF dataset

5.2.1 *Impact of input data size*

Results of experiments when varying the number of descriptors are shown in Figure 9 using a log scale on the x-axis. We omitted **H-BkNNJ** as it could not process the data in reasonable time. In Figure 9a, we can see that the execution time of the algorithms follows globally the same trend as with the Geo dataset, except for **PGBJ**. It is a computational intensive algorithm because the replication process implies calculating a lot of Euclidian distances. When in dimension 128, this part tends to dominate the overall computation time. Regarding disk usage (Figure 9b), **H-zkNNJ** is very high because we had to increase the number of shifted copies from 3 to 5 to improve the recall. Indeed, compared to the Geo dataset, it is very low (Figure 9c). Moreover, as the number of descriptors increases, **H-zkNNJ** goes from 30% to 15% recall. As explained before, the precision was found to be equal to the recall, which means the algorithm always returned k results. This, together with the improvement using more shifts, proves that the space filling curves using in **H-zkNNJ** are less efficient with high dimension data.

5.2.2 *Impact of k*

Figure 10 shows the impact of different values of k on the algorithms using a logarithmic scale on the x-axis. Again, since for **H-BNLJ** and **H-zkNNJ**, the complexity of the sorting phase is dependent on k, we can observe a corresponding increase of the execution time (Figure 10a). For **RankReduce**, the time varies a lot depending on k. This is because of the stochastic nature of the projection used in LSH. It can lead to buckets containing different number of elements, creating a load imbalance and some values of k naturally lead to a better load balancing. **PGBJ** is very dependent on the value of k because of the grouping phase. Neighboring cells are added until there are enough elements to eventually identify the k nearest neighbors. As a consequence, a large k will lead to larger group of cells and increase the computing time.

Figure 10b shows the effect of k on disk usage. **H-zkNNJ** starts with a very high ratio of 74 (not showed on the Figure) and quickly reduces to more acceptable values. **RankReduce** also experiences a similar pattern to a lesser extend. As opposed to the Geo dataset, SURF descriptors cannot be efficiently compressed, leading to large intermediate files.

Finally, Figure 10c shows the effect of k on the recall. As k increases, the recall and precision of **RankReduce** decreases for the same reason as with the Geo dataset. Also, for large k, the recall becomes lower than the precision because we get less than k results. The precision of **H-zkNNJ** decreases but eventually shows an upward trend. The increased number of requested neighbors increases the number of preceding and succeeding points copied, slightly improving the recall.

5.2.3 *Communication Overhead*

With the SURF dataset, we get a very different behavior than with the Geo dataset. The shuffle phase of **PGBJ** is very costly (Figure 11a). This is an indication of large replications incurred by the large dimension of the data and a poor choice of pivots. When they are too close to each other, entire cells have to be replicated during the grouping phase.

For **RankReduce** the shuffle is decreased but stay important, essentially because of the replication factor of 5. Finally, the shifts of original data in **H-zkNNJ** lead to a large communication overhead.

Considering now k, we have the same behavior we observed with the Geo dataset. The only difference is **PGBJ** which now exhibits a large communication overhead (Figure 11b). This is again because of the choice of pivots and the grouping of the cells. However, this overhead remains constant, irrespective of k.

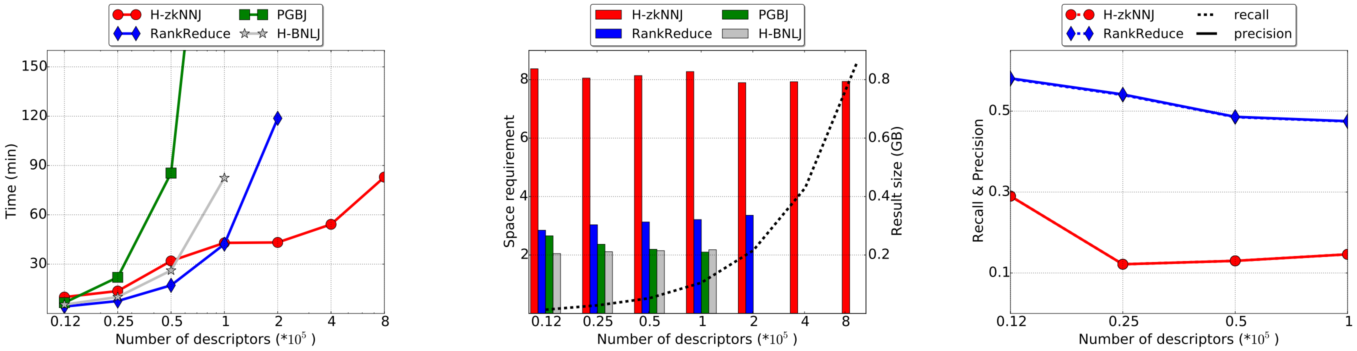
5.3 Impact of Dimension and Dataset

We now analyze the behavior of these algorithms according to the dimension of data. Since some algorithms are dataset dependent (i.e the spatial distribution of data has an impact on the outcome), we need to separate data distribution from the dimension. Hence, we use two different kinds of datasets for these experiments. First, we use real world data of various dimensions[10]. Second, we have built specific datasets by generating uniformly distributed data to limit the impact of clustering. All the experiments were performed using $0.5 * 10^5$ records and $k = 20$.

Since **H-BNLJ** relies on the dot product, it is not dataset dependent and its execution time increases with the dimension as seen on Figures 12a and 13a.

PGBJ is heavily dependent on data distribution and on the choice of pivots to build clusters of equivalent size which improves parallelism. The comparison of execution times for the datasets *128-sift* and *281-blog* in Figure 12a shows

[10] archive.ics.uci.edu/ml/datasets.html

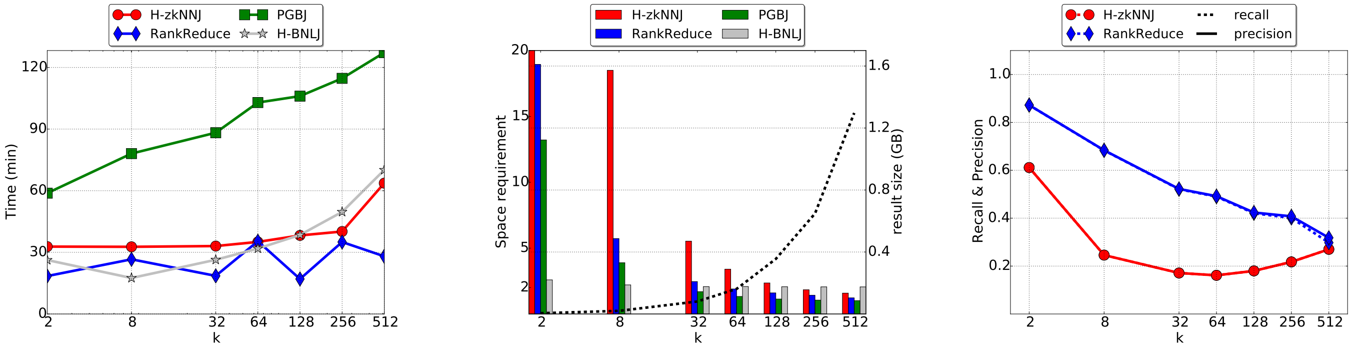


(a) Time

(b) Result size and Disk Usage

(c) Recall and Precision

Fig. 9: Surf, impact of the dataset size

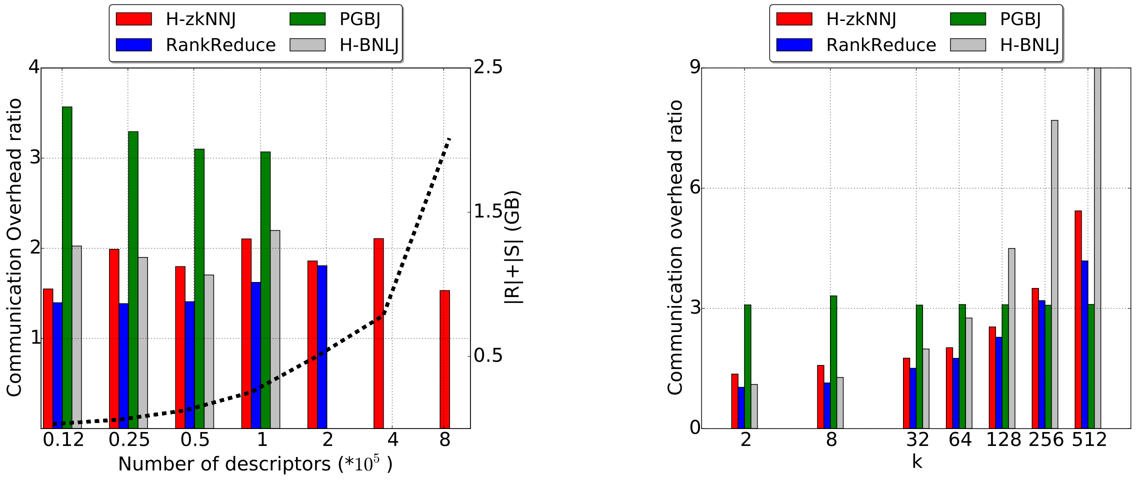


(a) Time

(b) Result size and Disk Usage

(c) Recall and Precision

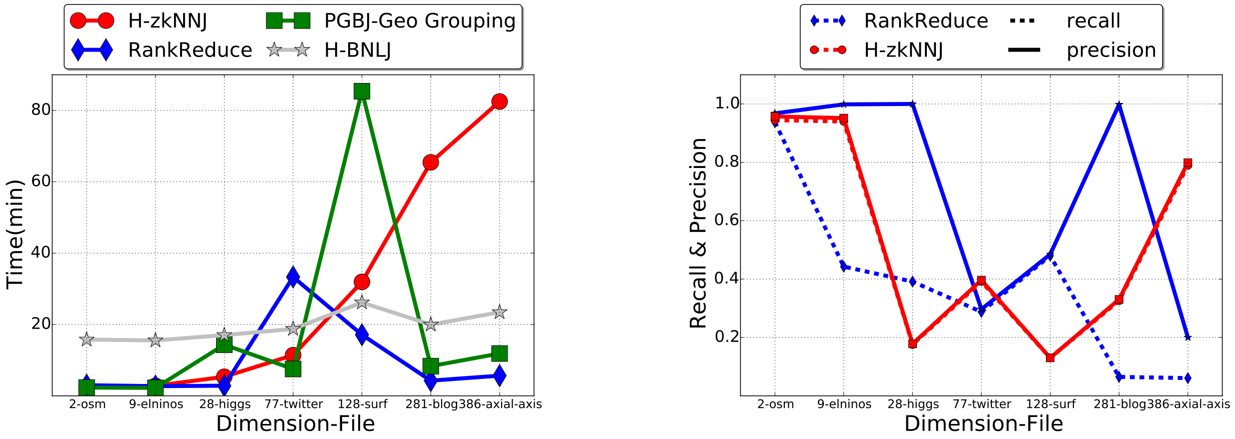
Fig. 10: Surf dataset with 50k records, impact of K,



(a) Surf, impact of the dataset size

(b) Surf dataset with 50k records, impact of K,

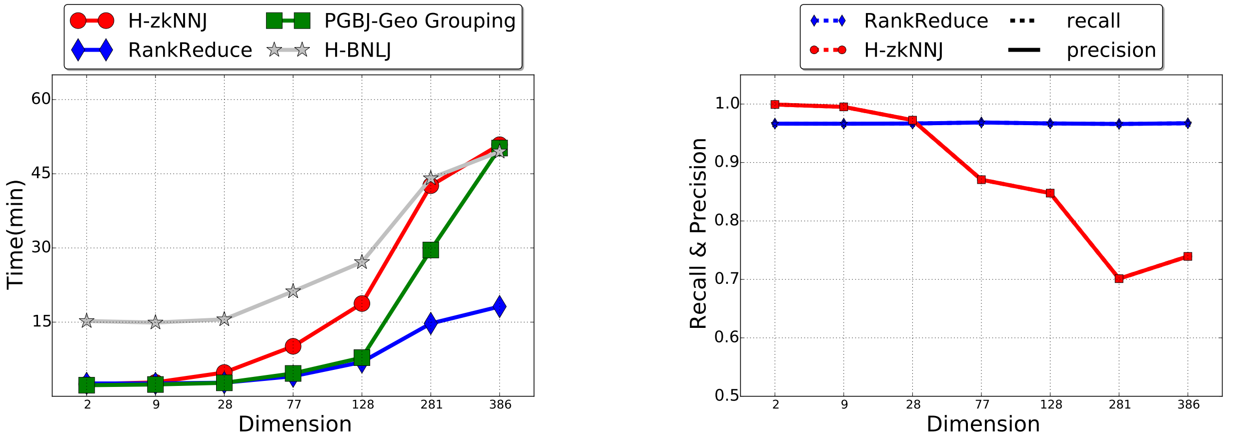
Fig. 11: Communication overhead for the Surf dataset



(a) Execution time

(b) Recall and Precision

Fig. 12: Real datasets of various dimensions



(a) Execution time

(b) Recall and Precision

Fig. 13: Generated datasets of various dimensions

that, although the dimension of data increases, the execution time is greatly reduced. Nonetheless, the clustering phase of the algorithm performs a lot of dot product operations which makes it dependent on the dimension, as can be seen in Figure 13a.

H-zkNNJ is an algorithm that depends on spatial dimension. Very efficient for low dimension, its execution time increases with the dimension (Figure 13a). A closer analysis shows that all phases see their execution time increase. However, the overall time is dominated by the first phase (generation of shifted copies and partitioning) whose time complexity sharply increases with dimension. Data distribution has an impact on the recall which gets much lower than the precision for some datasets (Figure 12b). With generated dataset (Figure 13b), both recall and precision are identical and initially very high. However as dimension increases, the recall decreases because of the projection.

Finally, **RankReduce** is both dependent on the dimension and distribution of data. Experiments with the real datasets have proved to be difficult because of the various parameters of the algorithm to obtain the requested number of neighbors without dramatically increasing the execution time (see discussion in Section 5.4.5). Despite our efforts, the precision was very low for some datasets, in particular *28-higgs*. Using the generated datasets, we see that its execution time increases with the dimension (Figure 13a) but its recall remains stable (Figure 13b).

5.4 Practical Analysis

In this section, we analyze the algorithms from a practical point of view, outlying their sensitivity to the dataset, the environment or some internal parameters.

5.4.1 H-BkNNJ

The main drawback of **H-BkNNJ** is that only the Map phase is in parallel. In addition, the optimal parallelization is subtle to achieve because the optimal number of nodes to use is defined by $\frac{input\ size}{input\ split\ size}$. This algorithm is clearly not suitable for large datasets but because of its simplicity, it can, nonetheless, be used when the amount of data is small.

5.4.2 H-BNLJ

In **H-BNLJ**, both the Map and Reduce phases are in parallel, but the optimal number of tasks is difficult to find. Given a number of partitions n, there will be n^2 tasks. Intuitively, one would choose a number of tasks that is a multiple of the number of processing units. The issue with this strategy is that the distribution of the partitions might be unbalanced. Figure 14 shows an experiment with 6 partitions and $6^2 = 36$ tasks, each executed on a reducer. Some reducers will have more elements to process than others, slowing the computation.

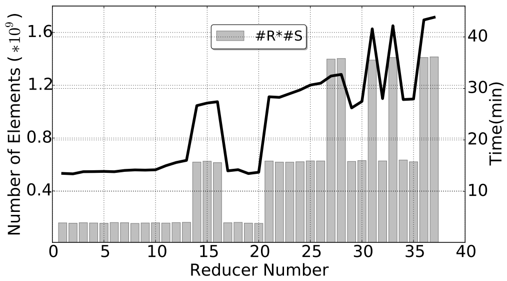


Fig. 14: H-BNLJ, candidates job, 10^5 records , 6 partitions, Geo dataset

Overall, the challenge with this algorithm is to find the optimal number of partitions for a given dataset.

5.4.3 PGBJ

A difficulty in **PGBJ** comes from its sampling-based preprocessing techniques because it impacts the partitioning and thus the load balancing. This raises many challenges. First, how to choose the pivots from the initial dataset. The three techniques proposed by the authors, farthest, k-means and random, lead to different pivots and different partitions and possibly different executions. We found that with our datasets, both k-means and random techniques gave the best performance. Second, the number of pivots is also important because it will impact the number of partitions. A too small or too large number of pivots will decrease performance. Finally, another important parameter is the grouping strategy used (Section 4.1). In Figure 15, we can see that the greedy grouping technique has a higher grouping time (bars) than the geo grouping technique. However, the global computing time (line) using this technique is shorter thanks to the good load balancing. This is illustrated by Figure 16 which shows the distribution of elements processed by reducers when using geo grouping (16a) or greedy grouping (16b).

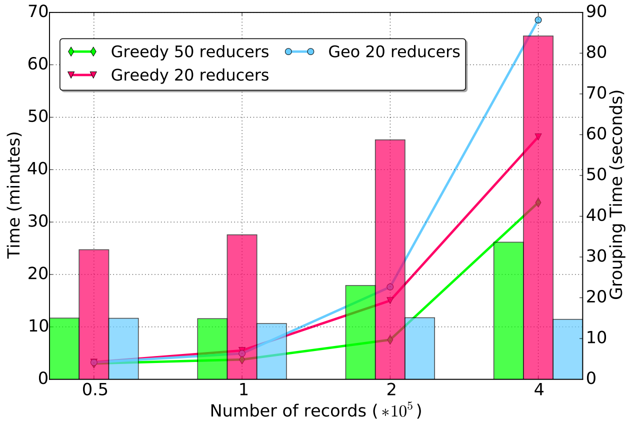
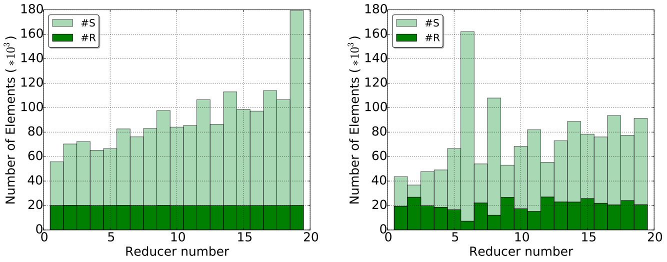


Fig. 15: PGBJ, overall time (lines) and Grouping time (bars) with Geo dataset, 3000 pivots, KMeans Sampling



(a) Geo

(b) Greedy

Fig. 16: PGBJ, load balancing with 20 reducers

5.4.4 H-zkNNJ

In **H-zkNNJ**, the z-value transformation leads to information loss. The recall of this algorithm is influenced by the nature, the dimension and the size of the input data. More specifically, this algorithm becomes biased if the distance between initial data is very scattered, and the more input data or the higher the dimension, the more difficult it is to draw the space filling curve. To improve the recall, the authors propose to create duplicates in the original dataset by shifting data. This greatly increases the amount of data to process and has a significant impact on the execution time.

5.4.5 RankReduce

RankReduce, with the addition of a third job, can have the best performance of all, provided that it is started with the optimal parameters. The most important ones are W, the size of each bucket, L, the number of hash families and M, the number of hash functions in each family. Since they are dependent on the dataset, experiments are needed to precisely tune them. In [38], the authors suggests this can be achieved with a sample dataset and a theoretical model. The first important metric to consider is the number of candidates available in each bucket. Indeed, with some poorly chosen parameter values, it is possible to have less than k elements in each bucket, making it impossible to have enough elements at the end of the computation (there are less than k neighbors in the result). On the opposite, having too many candidates in each bucket will increase too much the execution time. To illustrate the complexity of the parameter tuning operation, we have run experiments on the Geo and SURF datasets. First, Figure 17 shows that, for the Geo dataset, increasing W improves the recall and the precision at the expense of the execution time, up to an optimal before decreasing. This can be explained by looking at the number of buckets for a given W. As W increases, each bucket contains more elements and thus their number decreases. As a consequence, the probability to have the correct k neighbors inside a bucket increases, which improves the recall. However, the computational load of each bucket also increases.

A similar pattern can be observed with the SURF dataset (Figure 18, left), where increasing W improves the recall

Algorithm	Advantage	Shortcoming	Typical Usecase
H-BkNNJ	Trivial to implement	1. Breaks very quickly 2. Optimal parallelism difficult to achieve a priori	Any tiny and low dimension dataset (∼ 25000 records)
H-BNLJ	Easy to implement	1. Slow 2. Very large communication overhead	Any small/medium dataset (∼ 100000 records)
PGBJ	1. Exact solution 2. Lowest disk usage 3. No impact on communication overhead with the increase of k	1. Cannot finish in reasonable time for large dataset 2. Poor performance for high dimension data 3. Large communication overhead 4. Performance highly depends on the quality of a priori chosen pivots	1. Medium/large dataset for low/medium dimension 2. Exact results
H-zkNNJ	1. Fast 2. Does not require a priori parameter tuning 3. More precise for large k 4. Always give the right number of k	1. High disk usage 2. Slow for large dimension 3. Very high space requirement ratio for small values of k	1. Large dataset of small dimension 2. High values of k 3. Approximate results
RankReduce	1. Fast 2. Low footprint on disk usage	1. Fine parameter tuning required with experimental set up 2. Multiple hash functions needed for acceptable recall 3. Different quality metrics to consider (recall + precision)	1. Large dataset of any dimension 2. Approximate results 3. Room for parameter tuning

TABLE 4: Summary table for each algorithm in practice

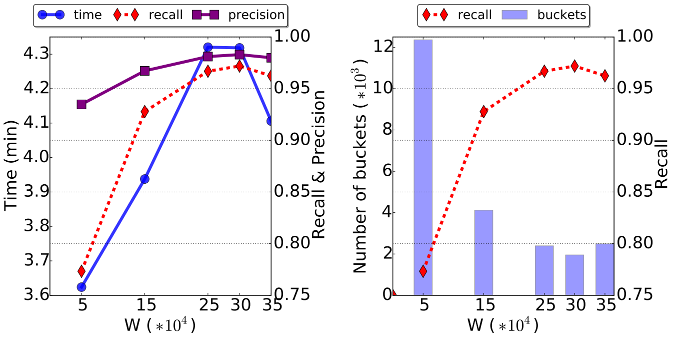


Fig. 17: LSH tuning, Geo dataset, 40k records, 20 nodes

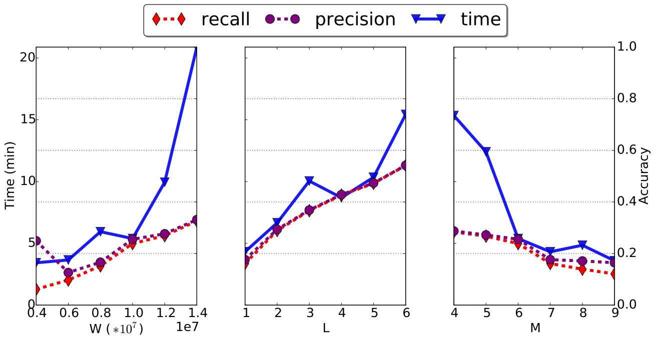


Fig. 18: LSH tuning, SURF dataset, 40k records, 20 nodes

(from 5% to 35%) and the precision (from 22% to 35%). Increasing the number of families L greatly improves both the precision and recall. However, increasing M, the number of hash functions, decreases the number of collisions, reducing execution time but also the recall and precision. Overall, finding the optimal parameters for the LSH part is complex and has to be done for every dataset

After finishing all the experiments, we found that the execution time of all algorithms mostly follows the theoretical analysis presented in Section 4. However, as expected, the computationally intensive part, which could not be expressed analytically, has proved to be very sensitive to a lot of different factors. The dataset itself, through its dimension and the data distribution, but also the parameters of some of the pre-processing steps. The magnitude of this sensitivity and its impact on metrics such as recall and precision could not have been inferred without thorough experiments.

5.5 Lessons Learned

The first aspect is related to load balance. **H-BNLJ** actually cannot guarantee load balancing, because of the random method it uses to split data. For **PGBJ**, Greedy grouping gives a better load balance than Geo grouping, at the cost of an increased duration of the grouping phase. At the same time, our experiments also confirm that **H-zkNNJ** and **RankReduce**, which use size based partitioning strategies, have a very good load balance, with a very small deviation of the completion time of each task.

Regarding disk usage, generally speaking, **PGBJ** has the lowest disk space requirement, while **H-zkNNJ** has the largest for small k values. However, for large k, the space requirement of all algorithms becomes similar.

The communication overhead of **PGBJ** is very sensitive to the choice of pivots.

The data are another important aspect affecting the performance of the algorithms. As expected, all the algorithms' performance decreases as the dimension of data increases. However, what exceeded the prediction of the theoretical analysis is that the dimension is really a curse for **PGBJ** . Because of the cost of computing distances in the pre-processing phase, its performance becomes really poor, sometimes worse than **H-BNLJ**. **H-zkNNJ** also suffers from the dimension, which decreases its recall. However, the major impact comes from the distribution of data.

In addition, the overall performance is also sensitive to some specific parameters, especially for **RankReduce**. Its performance depends a lot on some parameter tuning, which requires extensive experiments.

Based on the experimental results, we summarize the advantages, disadvantages and suitable usage scenarios for each algorithm, in Table 4.

6 Conclusion

In this paper, we have studied existing solutions to perform the kNN operation in the context of MapReduce. We have first approached this problem from a workflow point of view. We have pointed out that all solutions follow three main steps to compute kNN over MapReduce, namely preprocessing of data, partitioning and actual computation. We have listed and explained the different algorithms which could be chosen for each step, and developed their pros and cons, in terms of load balancing, accuracy of results, and overall complexity. In a second part, we have performed extensive experiments to compare the performance, disk usage and accuracy of all these algorithms in the same environment. We have mainly used two real datasets, a geographic coordinates one (2 dimensions) and an image based one (SURF descriptors, 128 dimensions). For all algorithms, it was the first published experiment on such high dimensions. Moreover, we have performed a fine analysis, outlining, for each algorithm, the importance and difficulty of fine tuning some parameters to obtain the best performance.

Overall, this work gives a clear and detailed view of the current algorithms for processing kNN on MapReduce. It also clearly exhibits the limits of each of them in practice and shows precisely the context where they best perform. Above all, this paper can be seen as a guideline to help selecting the most appropriate method to perform the kNN join operation on MapReduce for a particular use case.

After this thorough analysis, we have found a number of limitations on existing solution which could be addressed in future work. First, besides **H-BkNNJ**, all methods need to replicate the original data to some extend. The number of replications, although necessary to improve precision, has a great impact on disk usage and communication overhead. Finding the optimal parameters to reduce this number is still an open issue. Second, the partitioning methods are all based on properties of R. However, one can expect R to vary as it represents the query set. The cost of re-partitioning is currently prohibitive so, for dynamic queries, better approaches might rely on properties of S. Finally, MapReduce, especially through its Hadoop implementation, is well suited for batch processing of static data. The efficiency of theses methods on data stream has yet to be investigated.

Acknowledgments

This work was partly funded by the French Government (National Research Agency, ANR) through the Investments for the Future Program reference #ANR-11-LABX-0031-01. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by INRIA and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] D. Li, Q. Chen, and C.-K. Tang, "Motion-aware knn laplacian for video matting," in *ICCV'13*, 2013.

[2] H.-P. Kriegel and T. Seidl, "Approximation-based similarity search for 3-D surface segments," *Geoinformatica*, 1998.

[3] X. Bai, R. Guerraoui, A.-M. Kermarrec, and V. Leroy, "Collaborative personalized top-k processing," *ACM Trans. Database Syst.*, 2011.

[4] D. Rafiei and A. Mendelzon, "Similarity-based queries for time series data," *SIGMOD Rec.*, 1997.

[5] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *Foundations of Data Organization and Algorithms*, 1993.

[6] K. Inthajak, C. Duanggate, B. Uyyanonvara, S. Makhanov, and S. Barman, "Medical image blob detection with feature stability and knn classification," in *Computer Science and Software Engineering*, 2011.

[7] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast nearest neighbor search in medical image databases," in *VLDB'96*, 1996.

[8] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b$^+$-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.

[9] C. Böhm and F. Krebs, "The k-nearest neighbour join: Turbo charging the kdd process," *Knowl. Inf. Syst.*, vol. 6, no. 6, pp. 728–749, Nov. 2004.

[10] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *VLDB'97*, 1997.

[11] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *GeoInformatica*, 2010.

[12] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, 1975.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, 2008.

[14] G. Song, J. Rochas, F. Huet, and F. Magoulès, "Solutions for Processing K Nearest Neighbor Joins for Massive Data on MapReduce," in *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, Mar. 2015.

[15] N. Bhatia and A. Vandana, "Survey of nearest neighbor techniques," *International Journal of Computer Science and Information Security*, 2010.

[16] L. Jiang, Z. Cai, D. Wang, and S. Jiang, "Survey of improving k-nearest-neighbor for classification," in *Fuzzy Systems and Knowledge Discovery*, 2007.

[17] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive b$^+$-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, 2005.

[18] C. Böhm and F. Krebs, "The k-nearest neighbour join: Turbo charging the kdd process," *Knowl. Inf. Syst.*, 2004.

[19] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *Geoinformatica*, 2010.

[20] M. I. Andreica and N. T. pus, "Sequential and mapreduce-based algorithms for constructing an in-place multidimensional quad-tree index for answering fixed-radius nearest neighbor queries," 2013.

[21] C. Ji, T. Dong, Y. Li, Y. Shen, K. Li, W. Qiu, W. Qu, and M. Guo, "Inverted grid-based knn query processing with mapreduce," in *Proceedings of the 2012 Seventh Grid Annual Conference*, ser. CHINAGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–32. [Online]. Available: http://dx.doi.org/10.1109/ChinaGrid.2012.19

[22] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB'99*, 1999.

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

[24] A. S. Arefin, C. Riveros, R. Berretta, and P. Moscato, "Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus," *PLOS ONE*, 2012.

[25] D. Novak and P. Zezula, "M-chord: A scalable distributed similarity search structure," in *Scalable Information Systems*, 2006.

[26] P. Haghani, S. Michel, and K. Aberer, "Lsh at large distributed knn search in high dimensions ," in *WebDB'08*, 2008.

[27] A. Stupar, S. Michel, and R. Schenkel, "Rankreduce - processing k-nearest neighbor queries on top of mapreduce," in *In LSDS-IR*, 2010.

[28] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, 2012.

[29] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Extending Database Technology*, 2012.

[30] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Symposium on Computational Geometry*, 2004.

[31] G. Song, Z. Meng, F. Huet, F. Magoulès, L. Yu, and X. Lin, "A hadoop mapreduce performance prediction method," in *HPCC'13*, 2013.

[32] X. Zhou, D. J. Abel, and D. Truffet, "Data partitioning for parallel spatial join processing," *Geoinformatica*, 1998.

[33] Q. Du and X. Li, "A novel knn join algorithms based on hilbert r-tree in mapreduce," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on*, 2013, pp. 417–420.

[34] B. Yao, F. Li, and P. Kumar, "K nearest neighbor queries and knn-joins in large relational databases (almost) for free," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, March 2010, pp. 4–15.

[35] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors [lecture notes]," *Signal Processing Magazine, IEEE*, vol. 25, no. 2, pp. 128–131, March 2008.

[36] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proc. VLDB Endow.*, 2010.

[37] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *ECCV'06*, 2006.

[38] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling lsh for performance tuning," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, ser. CIKM '08, New York, NY, USA, 2008, pp. 669–678.

Ge Song Ge Song received the B.Sc. degree in Applied Mathematics and the M.Sc. degree in Engineering Sciences from Beihang University, China, in 2010 and 2013 respectively. She is currently a Ph.D. candidate in the Department of Mathematics and Computer Science at CentraleSupélec, Université Paris-Saclay, France. Her research interests include data mining, big data processing, parallel computing and real-time processing.

Justine Rochas Justine Rochas is a computer science Ph.D. candidate at University Of Nice Sophia Antipolis. Her research interests include programming languages, programming models and middlewares for parallel and distributed computing.

Lea El Beze Lea El Beze was born in Nice, France, in 1992 and acquired Israeli citizenship in 2013. She received a M.S. degree in computer science and Big Data from the University of Nice Sophia Antipolis, in 2015. She is now working on Middleware for Big Data in the private sector. http://lea-elbeze.alwaysdata.net/

Fabrice Huet Fabrice Huet holds a M.Sc. in Distributed System and a Ph.D. in Computer Sciences from the University of Nice Sophia Antipolis. He is currently an Associate Professor at the Computer Science department of the University Nice Sophia Antipolis. His research interests include high performance distributed computing, peer-to-peer architectures and distributed processing of data streams.

Frederic Magoules Frédéric Magoulès received the B.Sc. degree in engineering sciences, the M.Sc. degree in applied mathematics, the M.Sc. degree in numerical analysis, and the Ph.D. degree in applied mathematics from Université Pierre & Marie Curie, France, in 1993, 1994, 1995, and 2000, respectively. He is currently a Professor in the Department of Mathematics and in the Department of Computer Science at CentraleSupélec, Université Paris-Saclay, France. His research interests include parallel computing and data mining. Prof. Magoulès is a Fellow of IMA, a Fellow of BCS, a member of ACM, a member of SIAM and a member of IEEE.