K Nearest Neighbour Joins for Big Data on MapReduce: a Theoretical and Experimental Analysis [Experiments and Analysis Paper]

Ge Song[1,2], Justine Rochas[1], Lea El Beze[1], Fabrice Huet[1], and Frederic Magoules[2]

[1]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, France
justine.rochas,fabrice.huet@unice.fr lea.elbeze@gmail.com
[2]Ecole Centrale Paris, France
ge.song,frederic.magoules@ecp.fr

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.

I. Introduction

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

[TODO:Add some applications for [R3C2]—-Need to be checked] It is frequently used as a classification or clustering method in machine learning or data mining. The primary application of kNN join is k-nearest neighbour classification. Here 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 evaluated to estimate its class membership. This process can be considered as a kNN join of the testing set with the training set. For the well-known k-Means and k-Medoid clustering method, each of its iterations can be considered as a 1-NN join of the unclassified points with the center points. It can also be used to do density estimation. In particular, almost all stages of knowledge discovery process can be accelerated by using kNN join as a primitive operation like data preprocessing or data cleaning [1]. kNN join together with other methods based on it can be applied to a large number of fields, such as multimedia [2], [3], social network [4], time series analysis [5], [6], bio-information and medical imagery [7], [8].

[TODO:Je pense qu'il voudrait vraiment un example comme :] [TODO:Let an example to get a vision of this algorithm : Mr Bob wants rent a studio but he needs a school no far. So let R all studio to rent and S all the school. Mr Bob could be get the list of k nearest school for each studio.] [TODO:End of applications for [R3C2]]

[TODO:Add some challenges for [R3C1]—-Need to be checked] 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 on 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 $O(d \times |R| \times |S|)$. Then, finding the k nearest neighbours in S for every r in R boils down to sorting the computed 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 kNN computational complexity [9]–[13]. 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 [14] 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.

- 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.

[TODO:End of challenges for [R3C1]]

In this paper, we survey existing methods of kNN in MapReduce, focusing on the different steps involved in a computation, and give a thorough insight of their performance. It is a major extension of one of our previously published paper, [15], which provided only a simple theoretical analysis. Other surveys about kNN have been conducted, such as [16], [17], but they pursue a different goal. Indeed, in [16], the authors only focus on centralized solutions to optimize kNN computation whereas we target distributed solutions. In [17], 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. 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 III.
- A theoretical comparison of existing techniques in Section IV, focusing on load balancing, accuracy and complexity aspects.
- A complete implementation of 5 published algorithms and an extensive set of experiments using both low and high dimension datasets (Section V).
- 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.

II. Context

A. 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.

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($R \ltimes S$).

B. MapReduce

MapReduce [14] 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 [1] which provides a distributed platform for executing MapReduce jobs.

C. Existing kNN Computing Systems

The basic solution to compute kNN adopts a block 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

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

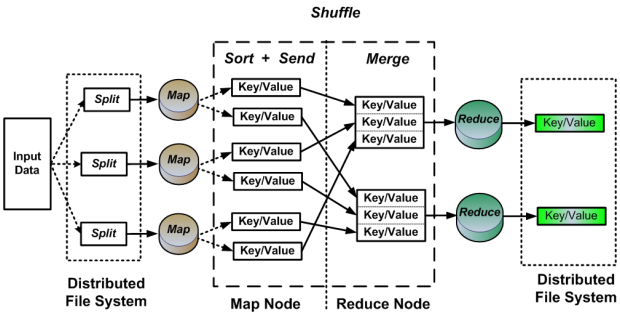


Fig. 1: Map Reduce Paradigm

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 [18] to index distances; [19] adopts a multipage overlapping index structure R-Tree; [11] proposes to use a balanced and dynamic M-Tree to organize the dataset; [20] introduces a sphere-tree with a sphere-shaped minimum bound to reduce the number of areas to be searched; [21] presents a multidimensional quad-tree in order to be able to handle large amount of data; [13] develops a kd-tree which is a clipping partition method to separate the search space; **[TODO:[R2C1] paper [2] — SOPHIE—Not Checked]**and [22] introduces a loose coupling and shared nothing 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) [23] and Space Filling Curve [24].

But with the increasing amount of data, these methods still can not handle kNN computation on a single machine efficiently. Experiments in [25] 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 [26], [27]. But most of them use the MapReduce model as it is naturally adapted for distributed computation, like in [28]–[30]. 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.

III. 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** [30]; A kNN based on Voronoi diagrams named **PGBJ** [29] and (2) Approximate solutions: A kNN based on z-value (a space filling curve method) named **H-zkNNJ** [30]; A kNN based on LSH, named **RankReduce** [28].

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.

A. 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 [30] 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) [23], [31]. 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: **[TODO:This part is added for explaining the accuracy of LSH later]**

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

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

where d(x, y) is the distance between two points x and y, and $d_1 < d_2$, $p_1 > p_2$. **[TODO:End]**

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 functions 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 [28] 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 [29] 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 next section. However, the reference PGBJ paper [29] only focuses on one pivot selection and one grouping strategies for evaluating the effect of k, of dimension, and of scale. We will experimentally show after that other strategies are also suitable for our dataset.

B. 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 [32]. 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 [33]. 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 shared-nothing partitions, while preserving locality information, have been proposed. They are described in the two next sections.

1) Distance Based Partitioning Strategy: The first partitioning strategy is based on Voronoi diagram, a method to divide the space into disjoint cells. The main property of this method 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 [29] gives a method to partition datasets R and S using Voronoi diagram. The partitioning principles are illustrated in Figure 2. After having identified the pivots p_i

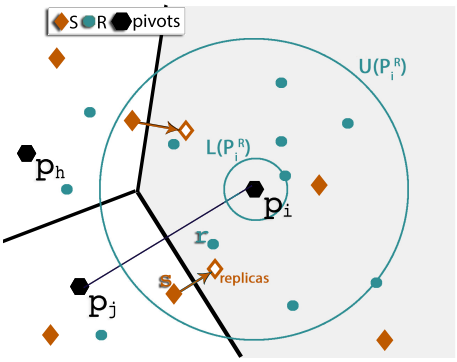


Fig. 2: Voronoi : cells partitioning and replication for cell P_i

in R (c.f. Section III-A), 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 (resp. P_i^S) from the pivot p_i. The boundaries and others statistics are used to find candidate data from S in the neighbouring cells. These data are then replicated in cell P_i^R. For example, in Figure 2, the element s of P_j^S falls inside $U(P_i^R)$ and is thus copied to P_i^R as a potential candidate for the kNN of r.

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 IV-A.

2) Size Based Partitioning Strategy: Another type of partitioning strategy aims at dividing data into equal size partitions. Paper [30] 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 the n quantiles. These quantiles are an unbiased estimation of the bounds of each partition. Figure 3 shows the partition bounds arising from the z-value method, and highlights i, a shifted data. **[TODO:check]** As said before, data are shifted in order to increase the result accuracy. After the shifted data being projected with the z-value space filling curve method (the 'Z' are visible thanks to the dotted line on Figure 3), data can be ordered in a dataspace of one dimension, represented on Figure 3 by dataspace lines Z_i^R and Z_i^S. Thanks to the previous sampling estimation, the dataspace lines can be divided into partitions. For a given partition of R_i, its kNN candidates from S_i are looked up in the dataspace line Z_i^S. For this, the k^th preceding and the k^th succeeding points of S_i are copied in in the bounds of the given partition of R_i. For example on Figure 3, four points of S_i are copied in partition 2, because they are considered as candidates for the kNN of the points of R_i in partition 2. **[TODO:end check]**

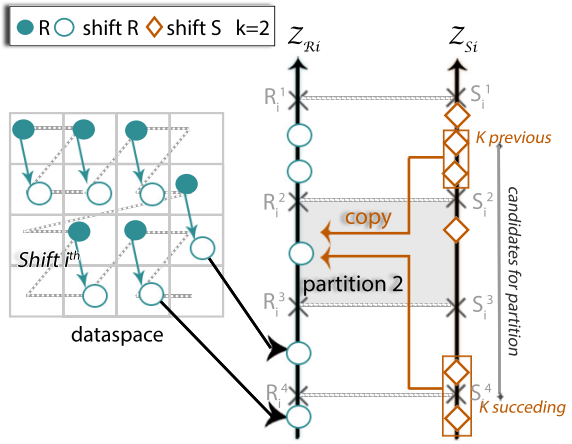


Fig. 3: z-value : partition

This method is likely to produce a substantially equivalent number of objects in each partition, thus naturally achieving load balancing. **[TODO:check]** Another similar size based partitioning method can be applied for data sets that are preprocessed with locality sensitive hashing, as illustrated in Figure 4. In the given example, there are 2 hashing families $a1$, $a2$, which means that each data will be hashed two times with two different hash functions. Each hashed data is then projected in the corresponding bucket. As said before, the principle of locality sensitive hashing is that it results in collisions for data that are spatially close to each other, which here allows data that are initially close to be hashed in the same bucket, provided that the bucket size (parameter w in LSH) is large enough to receive at least one copy of close data. **[TODO:end check]**

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, [29] uses a MapReduce job to perform the partitioning. In [30], both data preprocessing and data partitioning are completed in a single MapReduce job.

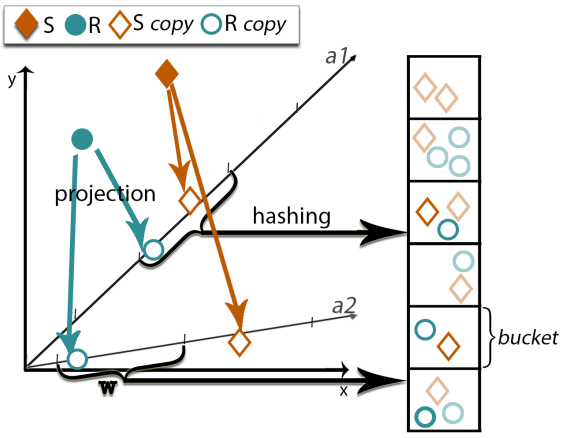


Fig. 4: LSH : bucket

C. 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.

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 [29] 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 $< R_i, S_i >$.

In RankReduce [28][2], the authors first preprocess data to reduce the dimentionality and partition data into buckets using LSH. Then, one MapReduce job is used to calculate the kNN join, with the Map phase taking each partition and the Reduce phase calculating and sorting the distances.

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.

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.

[TODO:[R2C1] Paper [1]—SOPHIE—Not Checked] Moreover, in order to avoid the scan of the whole dataset of each block, some index structures like R-Tree [30] or Hilbert R-Tree [34] can be used to index the local S blocks.

With preprocessing and partitioning strategies. In H-zkNNJ [30] 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, 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.

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

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

D. 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 5 summarizes the workflow we have gone through in this section and the techniques associated with each step.

IV. Theoretical Analysis

A. 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:

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

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 $\mathcal{O}(Task)$ 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

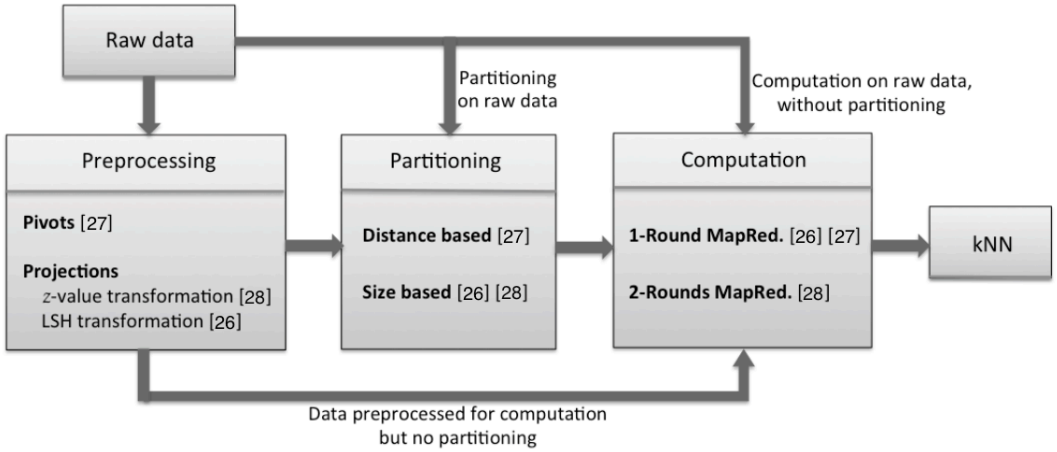


Fig. 5: Usual workflow of a kNN computation using MapReduce

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.

Finally, The H-zkNNJ method assumes:

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

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 [30], the authors give a short proof which shows that the worst-case 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

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

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.

This load balancing is our improvement and obtained significant results. **[TODO:it was not in the original paper?]**

B. Accuracy

[TODO:Add theoretical analyses about Accuracy—Need to be checked]

Usually, the lack of accuracy is the direct consequence of techniques to reduce the dimensionality with techniques such as z-values and LSH. In [30] (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 [35] 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 [36] 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 with a fixed bucket size W, this probability decreases as the distance d increases. Another way to improve the accuracy of LSH can be obtained by increasing the number of hash functions used. Unlike z-value, the performance of LSH depends a lot on parameter tuning.

[TODO:END]

C. Global Complexity

[TODO:T2—-Totally changed, need to be checked] When computing kNN in MapReduce, the theoretical complexity is not sufficient to estimate the execution time. But we can still list some additional factors strongly impact the execution time:

(1) **The number of MapReduce jobs:** Starting a job (whether in Hadoop [37] 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 distances to compute and to sort to find the final result for each object** r_i**:** Sorting is a computational intensive operation, so the number of elements to be sorted also impacts computation time. In fact, like the indexes method used in non-parallel methods to avoid the scan of the whole dataset, the technologies of pre-processing and partitioning used by each parallel method and the counterbalance of using multiple jobs are all in order to reduce the proportion of replicated data, thereby reducing the impact of this number on the final computation time.

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 Shuffle phase.
- Computation overhead, is mainly composed by two parts, 1). the computation of the distances 2). the sort of the distance. 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 distance 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.

Suppose we have a good load balance for each method, which means if we have n splits for R, the number of elements in one split R_i can be considered as $\frac{|R|}{n}$ ideally, and it stays the same for every task. The total sort complexity for one task is: $|R_i| \cdot \mathcal{O}(sort\ r_i) = \frac{|R|}{n} \cdot \mathcal{O}(sort\ r_i)$. The difference comes from the complexity of sort for one r_i, which is the number of distances need to be sorted for one r_i.

In order to improve the efficiency of sort, we use a priority queue for each r_i to select the top k nearest neighbor in each method.

The basic method H-BkNNJ only uses one MapReduce job, and requires only one Reduce task to calculate and sort the distances. The communication overhead is $\mathcal{O}(|R| + |S|)$. The complexity of sorting all distances for one r_i in R is $\mathcal{O}(|S| \cdot log\,|k|)$, and the total sort cost for one task is $\mathcal{O}(|R| \cdot |S| \cdot \log|k|)$. Since R and S are usually large dataset, this method quickly becomes impracticable.

To overcome this limitation, H-BNLJ [30] uses two MapReduce jobs, with n^2 tasks to compute the distances. The total communication overhead is $\mathcal{O}(n\,|R| + n\,|S| + kn\,|R|)$. However, using a second job significantly reduces the complexity of sorting for each r_i to $(n \cdot k) \cdot log\,(k)$, with k the number of nearest neighbors required. 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 [29] performs a preprocessing phase followed by two MapReduce jobs. This method also only uses n Map tasks to calculate the distances. Overall, the sorting complexity is reduced to $\mathcal{O}(|S_i| \cdot log\,|k|)$ for each r_i, and $\mathcal{O}(\frac{|R|}{n} \cdot |S_i| \cdot log\,|S_i|)$ totally for one task. And the shuffling cost is $\mathcal{O}(|R| + |S| + |RepS_c| \cdot n)$. In the original paper the authors gives a formulation for calculating $|RepS_c| \cdot n$ which is called the number of replicas of objects in S represented by $RP(S)$.

In RankReduce [28], the initial dataset is projected by L hash functions **[TODO:famillies?]** into buckets. Each bucket contains a piece of R and a piece of S. After finding the local candidates by the second job, the third job needs to combines the local results in order to find the global k nearest neighbor. The sort complexity for one r_i becomes $\mathcal{O}(L \cdot k \cdot log(k))$, and $\mathcal{O}(|R_i| \cdot L \cdot k \cdot log(k))$ totally for one task. And the total communication cost becomes $\mathcal{O}(L \cdot (|R| + |S|) + L \cdot k \cdot |R|)$.

H-zkNNJ [30] also begins by a preprocessing phase and uses in total three MapReduce jobs in exchange for taking only n Map tasks to compute the distances. Moreover, they only take $(r_i, C_i\,(r_i))$, that is the candidate neighbor set, into account. Since $C_i\,(r_i)$ contains only $2k$ neighbors for each r_i, the complexity is now reduced to $\mathcal{O}((k) \cdot log\,(2 \cdot k))$ for one shift of data, and $\mathcal{O}(\frac{|R|}{n} \cdot (2 \cdot \alpha \cdot k) \cdot log\,(k))$ in total per task for α shifts of data. The communication overhead is $\mathcal{O}(\frac{1}{\varepsilon^2} + |S| + k \cdot |R|)$, with $\varepsilon \in]0, 1[$.

Besides the complexity of the algorithms, the dimension of data will also strongly affect the running time. When the dimension of data increases, we will inevitably encounter the curse of dimensionality. Therefore, for high dimensional data, despite the fact that RankReduce and H-zkNNJ use more MapReduce jobs, they should have a shorter running time than the others. This is due to the initial reduction of the dimension.

Since we have analyzed these 3 important factors that affect the performance of each algorithm, we may have some questions like:

- Is it possible to find a tradeoff among these three factors to minimize the total processing time of each method?
- Is it possible to find a formula about the overall complexity represented by these three factors.

Unfortunately, the answers are no. Because, firstly, in real applications, the distribution of data also plays tremendous impacts on the running time of each method, and this impact can not be qualified analysis; Then, the overall complexity is not a simple combination of these three factors. For example, to reduce the number of the candidates for the final sorts, H-

BNLJ, RankReduce and H-zkNNJ choose to pre-sort the local candidates in there previous MapReduce job, and the total complexity should also consider about the complexity for this process. At the same time, the time spent for finding pivots for PGBJ, for hashing in RankReduce and for calculating z-value in H-zkNNJ as long as their partition time should also be considered; Finally, the number of nodes in the cluster, the capacity of each nodes and the communication among nodes can also affect the performance of each method.

We need to remind the readers that the overall complexity is not the summation or a simple combination of the complexity of each part. And the total processing time can not be predicated by theoretical analysis, which makes it more important to have an experimental study.

D. 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 **??** shows a summary of the systems we have examined and their main characteristics.

V. Evaluation

To validate our analysis, we performed an extensive experimental evaluation of the algorithms described in the previous sections. For H-zkNNJ and H-BNLJ, we took the code provided by the authors as a starting point[4] 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 reducer to the first phase to compute the statistics for each bucket. This statistics are used for load balancing. Using these statistics, a custom partitioner distributes buckets among the reducers to try to achieve good work balance. Moreover, in the published version, the authors performs only a single hashing of the data. To improve the accuracy, we perform multiple **[TODO:how many?]** hashes. As a consequence, our version of RankReduce uses three phases instead of two. **[TODO:this should be explained earlier. Also we should use it in the discussion about the impact of multiple phases]**.

Most of the experiments were ran using two different datasets: one in dimension 2, the *geographic - or Geo - dataset*, and one in dimension 128, the *image feature descriptors - or SURF - dataset*. Table I gives the number of records and the size of the various sets used in the experiments. Since in all our experiments we have $|R| = |S|$, note that we process twice the amount of data indicated in the table. This is a worst case scenario in terms of dataset size. As we will see in the following sections, it is difficult to process event modest file sizes of large dimension data.

records($\times 10^5$)	Geo	Surf
0.125	1.8MB	16MB
1	15MB	126MB
2	29MB	251MB
4	57MB	392MB
8	114MB	N/A
32	459MB	N/A
256	3.6GB	N/A

TABLE I: Size of R and S with the Geo and Surf datasets

For all experiments, we have set $k = 20$ unless specified otherwise.

The accuracy of approximate algorithms is computed against the exact algorithm PGBJ with the following ratio $\wp$, taken from [38]: $\wp = \frac{|A(v) \bigcap I(v)|}{|I(v)|}$ where $I(v)$ is the results of exact kNN of v and $A(v)$ the results of kNN given by the approximate methods. We compute the space requirement of each algorithm with $\frac{Size_{final}+Size_{intermediate}}{Size_{final-wanted}}$.

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. 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 8. 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 experiment using at most 20 nodes.

A. Geographic dataset

The Geo dataset involves geographic XML data in two dimensions[6]. For all experiments in this section, we used the parameters described in Table II. Details regarding each parameter can be found in sections III-A and III-B.

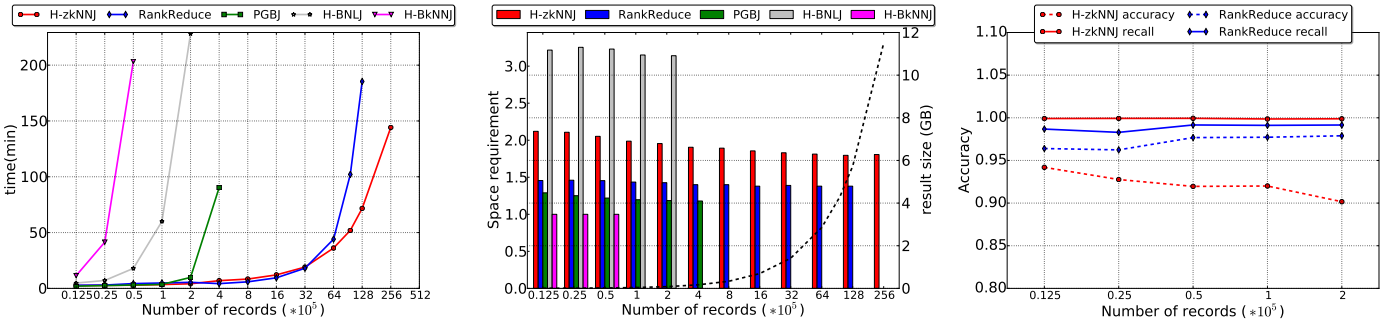
Algorithm	Partitioning	Reducers	Other
H-BNLJ	10 partitions	100 reducers	
PGBJ	3000 pivots	25 reducers	k-means + greedy
RankReduce	W = $32 * 10^5$	25 reducers	L = 2 M = 7
H-zkNNJ	10 partitions	30 reducers	3 shifts, p=10

TABLE II: Algorithm parameters for geographic dataset

[4]http://ww2.cs.fsu.edu/~czhang/knnjedbt/

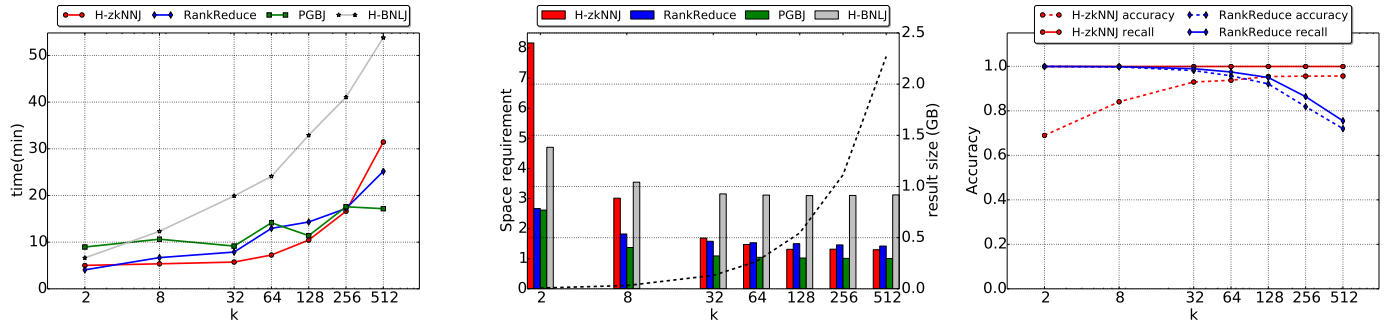
[5]www.grid5000.fr

[6]Taken from www.download.geofabrik.de



(a) Time (b) Result size and Disk Usage (c) Accuracy and recall

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



(a) Time (b) Result size and Disk Usage (c) Accuracy and recall

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

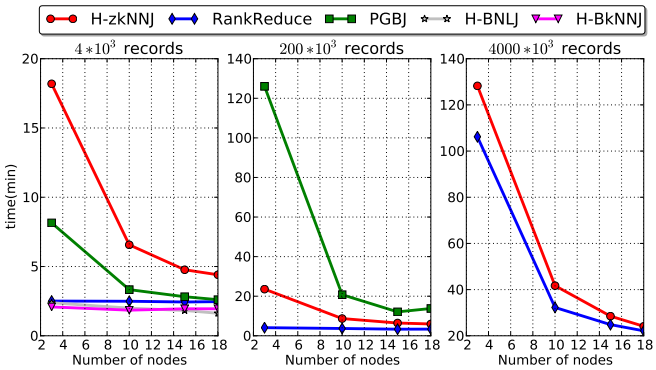


Fig. 8: Impact of the number of nodes on computing time

1) Impact of input data size: Our first set of experiments measure the impact of the data size on execution time, disk space and accuracy.

Figure 6a shows the global computing time of all algorithms, varying the number of records from $0.1 \cdot 10^5$ to $256 \cdot 10^5$. The global computing time increases more or less exponentially for all algorithms, but only H-zkNNJ and RankReduce can process medium to large dataset. For small datasets, PGBJ can compute an exact solution as fast as the other algorithms.

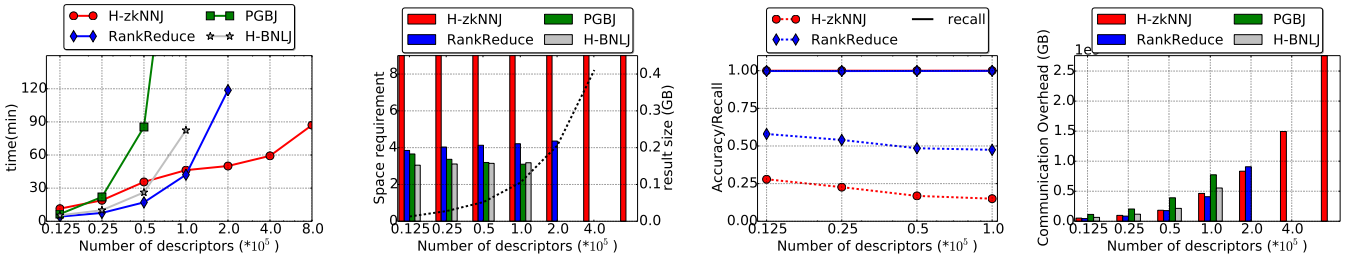
Figure 6b shows the space requirement of each algorithm as a function of the final output size. To reduce the footprint of each run, intermediate data are compressed. For example, for H-BNLJ, the size of intermediate data is 2.6 times bigger than the size of output data. Overall, the algorithms with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

Figure 6c shows the accuracy of the two approximate algorithms, H-zkNNJ and RankReduce. As the number of records increases, the accuracy of H-zkNNJ starts to decrease, while still being high, because of the space filling curves used in the preprocessing phase. On the other hand, RankReduce benefits from larger datasets because more data end up in the same bucket, increasing the number of candidates.

2) Impact of k: The number of requested nearest neighbours, k, can have a significant impact on the performance of some of the kNN algorithms. We experimented on a dataset of $2 \cdot 10^5$ records, except for H-BNLJ where only $5 \cdot 10^4$ records were used for performance reasons. We used values for k varying from 2 to 512 and measured computing time, disk usage, and accuracy, as shown in Figures 7, using a logarithmic scale on the x-axis.

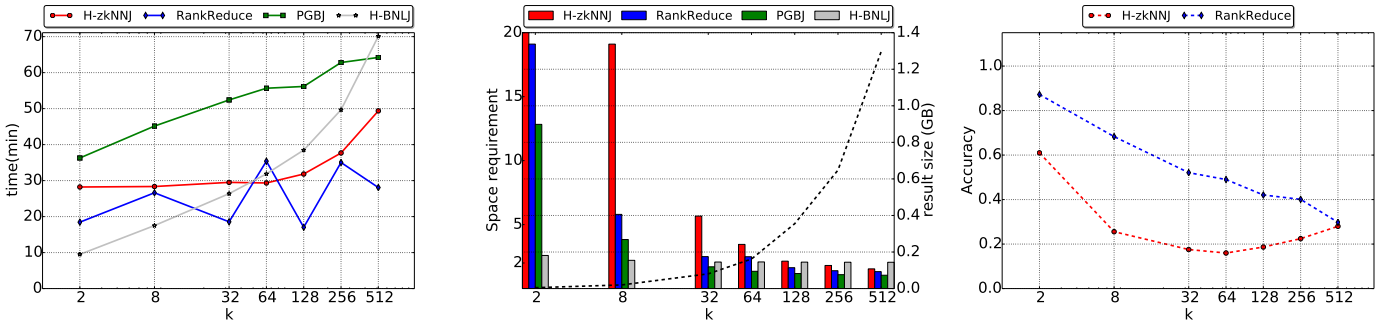
Firstly, we observe a global increase in computing time (Figure 7a) which matches the complexity analysis performed earlier. As k increases, the performance of H-zkNNJ, compared to the other advanced algorithms, decreases. This is due to the necessary replication of the z-values of S throughout the partitions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

Secondly, 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 quicker than the intermediate data size. As a consequence, there is no particular algorithm that suffers from such a bottleneck at this point. PGBJ is nevertheless



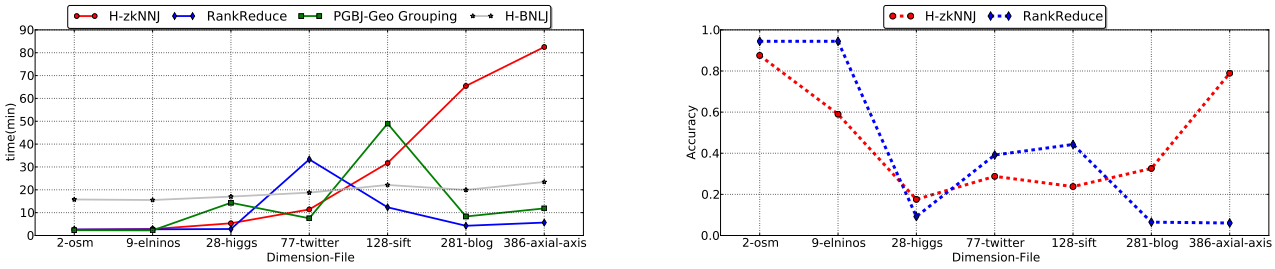
(a) Time (b) Result size and Disk Usage (c) Accuracy (d) Communication overhead

Fig. 9: Surf, impact of the dataset size



(a) Time (b) Result size and Disk Usage (c) Accuracy

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



(a) Execution time (b) Accuracy

Fig. 11: real datasets of various dimensions

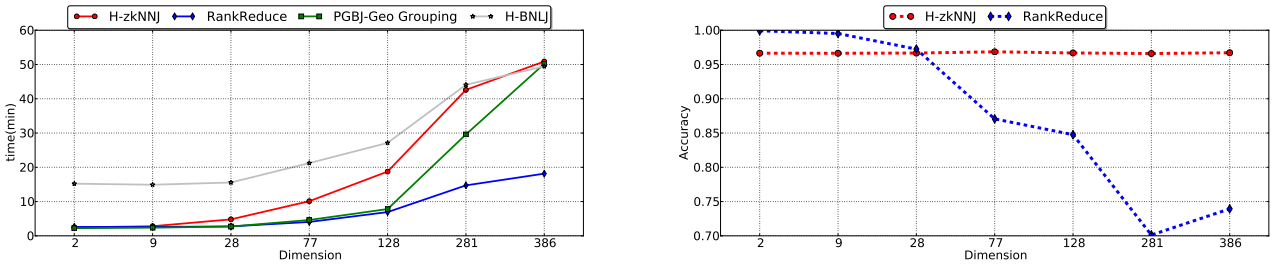
the most efficient from this aspect. Its replication of data occurs independently of the number of selected neighbors. Thus, increasing k has a small impact on this algorithm, both in computing time and space requirements. On this figure, an interesting observation can also be made for H-zkNNJ: the first value of k shows an explosion of intermediate data, whereas the ratio is much more balanced for the next values of k. This is due to the fact that H-zkNNJ creates several copies of the initial dataset in order to improve the accuracy. Consequently, as the number of copies is the same regardless of the value of k, this algorithm is more efficient considering disk usage when k is large.

Surprisingly, changing k has a different impact on the accuracy of the approximate kNN methods, as can be seen on figure 7c. For RankReduce, increasing k has a negative impact on the accuracy which sharply declines when $k \geq 64$. The reason of this drop is that the window parameter of LSH remains unchanged for all the experimented k. The window parameter was empirically set for the first value of k we experimented with. So it was, at first, well-adapted but became less optimal as k increased. This shows there is a link between global parameters such as k and parameters of the LSH process. Finally, when using H-zkNNJ, increasing k stabilizes the accuracy: the probability to have incorrect points is reduced as there are more candidates of kNN points in a single partition.

B. *Image Feature Descriptors dataset*

We now investigate whether the dimension of input data has an impact on the kNN algorithms. Image feature descriptors are known to be high dimensional data, so we chose to use SURF [39], which produces data in dimension 128. Using a public set of images[7], we have computed the SURF descriptors and used them for the kNN computation. We used the Euclidian distance between descriptors to measure image similarity.

[7] www.vision.caltech.edu/Image_Datasets/Caltech101



(a) Execution time

(b) Accuracy

Fig. 12: Generated datasets of various dimensions

For all experiments in this section, the parameters mentioned in Table III are used.

Algorithm	Partitioning	Reducers	Other
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	10 partitions	30 reducers	5 shifts

TABLE III: Algorithm parameters for SURF dataset

Results of experiments when varying 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 **??**, 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, H-zkNNJ is very high because we had to increase the number of shifted copies from 3 to 5 to improve the accuracy. Compared to the Geo dataset, the accuracy is very low and as the number of descriptors increases, H-zKNNJ goes from 30% to 15% of accuracy.

1) 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.

Figure 10b shows the effect of k on disk usage. H-zKNNJ starts with a very high ratio of 74 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 accuracy. As k increases, the accuracy of RankReduce decreases for the same reason as with the Geo dataset. On the other hand the accuracy of H-zkNNJ eventually shows an upward trend.

C. 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 differents datasets for these experiments. First, we use real world data of various dimensions[8]. Second, we have built specific datasets by generating uniformely distributed data to ensure the absence of clusters.

Dimension	Real	Synthetic
2	7.1MB	846KB
9	3.5MB	2.8MB
28	35MB	8MB
77	23MB	21MB
128	63MB	36MB
281	60MB	78MB
386	73MB	107MB

TABLE IV: Size of $0.5*10^5$ records for R and S in dimension experiments

For this set of experiments we used $k = 20$.

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 11a and 12a.

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 11a shows that the dimension supports this analysis. 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 12a when using the generated dataset.

H-zkNNj is an algorithm that depends on spatial dimension. Very efficient for low dimension, its execution time increases exponentially with the dimension (Figure 12a). 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 accuracy. Our generated dataset provides a constant accuracy (Figure 12b) whereas, with real world data, the accuracy varies a lot (Figure 11b).

Finally, RankReduce is both dependent on the dimension of data and on the dataset. First, using the generated datasets,

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

we see that its execution time increases with the dimension (Figure 12a) although its accuracy decreases (Figure 12b). Experiments with the real datasets have proved 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 V-D5). The accuracy is both dependent on the dimension and the dataset.

D. 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.

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 dependent on the size of an input split (the unit that will be processed by a Map). The optimal number of nodes to use is defined by $\frac{input\ size}{input\ split\ size}$. This algorithm is clearly not suitable for large dataset but because of its simplicity, it can, nonetheless, be used when the amount of data is small.

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 13 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.

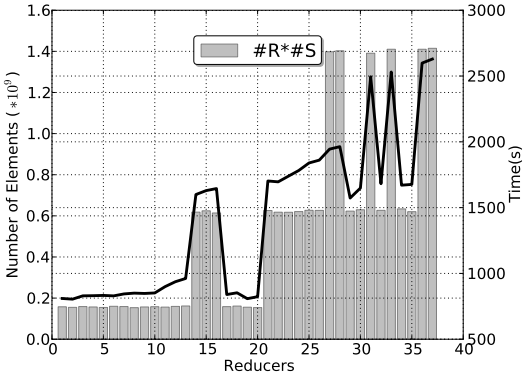


Fig. 13: H-BNLJ, candidates job, 10^5 records , 6 partitions

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

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 two main questions. 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 the Geo dataset, 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. Both a too small or too large number of pivots will decrease performance. Another important parameter is the grouping strategy used (Section IV-A). In Figure 14, we can see that

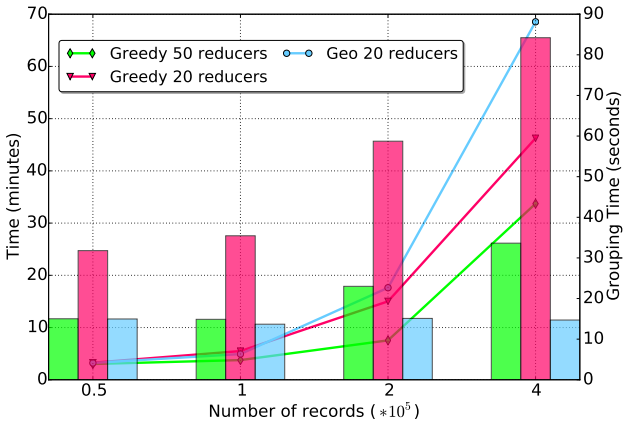
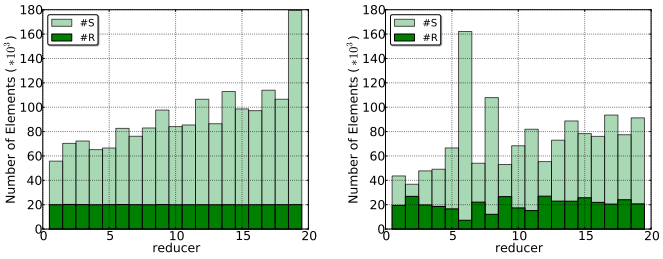


Fig. 14: PGBJ, overall time (lines) and Grouping time (bars) with Geo dataset, 3000 pivots, KMeans Sampling

the greedy grouping technique has a higher grouping time (bars) than the geo grouping technique. However, the global computing time using this technique is shorter thanks to the good load balancing. This is illustrated by Figure 15 which shows the distribution of elements processed by reducers when using geo grouping (15a) or greedy grouping (15b).



(a) Geo (b) Greedy

Fig. 15: PGBJ, load balancing with 20 reducers

4) H-zkNNJ: In H-zkNNJ, the z-value transformation leads to information loss. The accuracy of this algorithm is influenced by the nature of the dataset, 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 accuracy, 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) 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 related to W (size of each bucket), and to the hash

Algorithm	Advantage	Shortcoming	Typical Dataset
H-BkNNJ	Trivial to implement	Breaks very quickly	Any tiny dataset
H-BNLJ	Easy to implement	Slow, does not scale	Any small/medium dataset
PGBJ	1. Exact solution 2. Lowest disk usage	1. Cannot finish in reasonable time for large dataset 2. Bad performance for high dimension data	Medium/large dataset for low/medium dimension that requires exact results
H-zkNNJ	1. Fast 2. Can treat data on the fly	1. High disk usage 2. Bad accuracy for high dimension data	Large dataset for low/medium dimension that can accept approximate results
RankReduce	1. Fast 2. Low footprint on the disk usage	Require precise parameter tuning with experimental set up	Large dataset of any dimension that do not change often and can accept approximate results

TABLE V: Summary table for each algorithm in practice

functions, namely M (number of hash functions in a family) and L (number of families). 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 parameters, 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 16 shows that, for the Geo dataset, increasing W improves the accuracy and the recall at the expense of execution time, up to an optimal before decreasing. Low values for W lead to a large number of buckets **[TODO:What conclusion can we draw?]**.

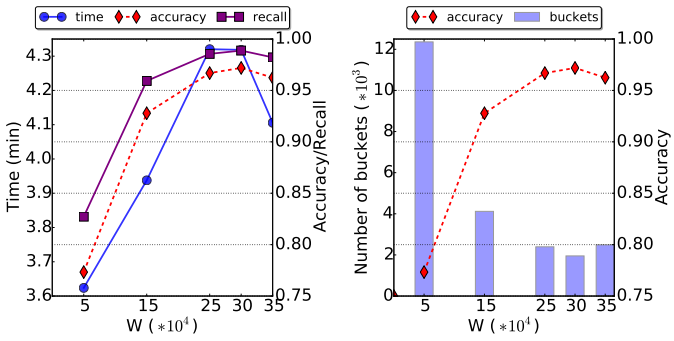


Fig. 16: LSH tuning, Geo dataset, 40k records, 20 nodes

A similar pattern can be observed with the SURF dataset (Figure 17, left), where increasing W improves the accuracy and the recall. However, the increase of the recall (from 22% to 95%) is much larger than the accuracy (from 5% to 35%). Increasing M decreases the number of collisions, reducing execution time but also possibly the accuracy. Finally, increasing L improves the accuracy but increases the execution time.

Overall, we found that finding optimal parameters for the LSH part is a very time consuming process which 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 IV. However, we found shortcomings or sensitivity to parameters which could not be inferred without experiments.

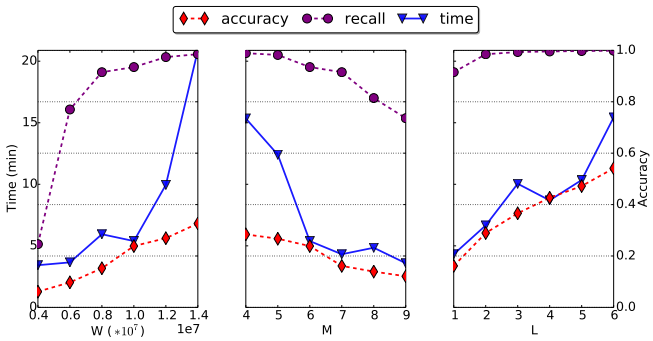


Fig. 17: LSH tuning, SURF dataset, 40k records, 20 nodes

E. 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. However, contrarily to our expectation, the intermediate space growth rate reduces as k increases.

The dimension of data is another important aspect affecting the performance of the algorithms. As expected, all the algorithms' performance decreases as the dimension of data increases. However, what exceeds the prediction of the theoretical analysis is that the dimension is really a curse for PGBJ. Because of the cost of computing distances, the performance of PGBJ becomes really poor, sometimes worse than H-BNLJ. H-zkNNJ also suffers from the dimension, its accuracy collapses to less than 20% in this case.

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

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

VI. 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. **[TODO: Thanks to these analysis, we can go further. For example the algorithm can be mixed together. By the experience, we demonstrated that 3MR(map reduce job) cost less 2 MR. If we cut each bucket like of new R and S and apply independant job, may be n MR would be better. And it integrates the idea of level mentionated in this papper http://gamma.cs.unc.edu/KNN/bilevel.pdf . Another point is the partitionning . Z value depends to S. Voronoi of statistic of R and of S. How can we partitionate just for the dataset S and do R arrive in the fly ?. Another problematic is the replication of data, Does it exist a way to avoid the replication. Maybe we can imagine a system of propagation on cells close ? Another point is about paradigm. Map Reduce is it more suitable or the technologies like Spark, Flink that uses ETL are better in this case ?]**

References

[1] C. Xia, H. Lu, B. C. Ooi, and J. Hu, "Gorder: An efficient method for KNN join processing," in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, 2004, pp. 756–767.

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

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

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

[22] 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 *ChinaGrid Annual Conference (ChinaGrid), 2012 Seventh*, Sept 2012, pp. 25–32.

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

[24] 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.

[25] 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.

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

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

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

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

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

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

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

[38] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling lsh for performance tuning," in *Information and Knowledge Management*, 2008.

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