Solutions for Processing K Nearest Neighbor Joins for Big Data on MapReduce

Ge Song*†, Justine Rochas*, Fabrice Huet* and Frederic Magoules†
*Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
†Ecole Centrale de Paris, France
ge.song@inria.fr, {justine.rochas,fabrice.huet}@unice.fr, frederic.magoules@ecp.fr

***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 increases, only distributed approaches can perform such costly operation in a reasonable time. Recent work has focused on implementing efficient solutions using the MapReduce programming model because it is suitable for large scale data processing. Also, it can easily be executed in a distributed environment. Although these works provide different solutions to the same problem, each one has particular constraints and properties. There is no readily available comparison to help users choose the one most appropriate for their needs. This is the problem we address in this work. Firstly, we show that all kNN implementations go through a common workflow, which we use as a basis for classification. Secondly, we describe precisely the different techniques published so far. And lastly, we provide a set of objective criteria, such as accuracy and complexity, which can be used to make informed decisions.**

***Index Terms*—kNN Join; Data Partition; Hadoop; MapReduce;**

I. Introduction

Given a set of query points R and a set of reference points S, a k nearest neighbor join (hereafter kNN) 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. It 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 is to perform a pairwise computation of distance for each element in R and each element in S. The computational complexity of this pairwise calculation is $O(|R| \times |S|)$. Then, finding the k nearest neighbors in S for every r in R boils down to sorting the computed distances, and leads to a complexity of at least $|S| \times log\,|S|$. 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 [8] [9] [10] [11] [12].

Although variety of previous work has greatly improved its performance, 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 especially designed for data-intensive processing. It was first introduced by Google [13] and popularized with the Hadoop [14] framework, an open source implementation. MapReduce offers a programming model adapted to large scale data processing, which can easily be distributed. The framework can be installed on commodity hardware and automatically distribute a MapReduce job over a set of machines. Therefore, it seems to be a good target for computing kNN in a distributed way. However, writing an efficient kNN in MapReduce is challenging. First, classical algorithms have to be redesigned to fit the MapReduce programming model. Second, data partition and distribution strategies have to be carefully designed to limit communications and data transfer. In this paper, we review existing implementations of kNN in MapReduce, focusing on different steps involved in a computation. Other surveys about kNN have been conducted, such as [15], [16]. 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 based on a theoretical performance analysis. Our approach is much more high level, though we also provide some details about the performance of the works we study. Overall, our contributions are:

- The decomposition of a distributed kNN computation in different steps.
- An analysis of existing techniques for each step based on load balancing, accuracy and complexity.
- A global comparison which can be used to find the most suitable solution for a given use case.

The rest of the paper is organized as follows. Section II introduces the background about kNN and MapReduce. Section III presents the 3 stages of workflow involved in computing kNN on MapReduce. Section IV analyzes existing implementations from the following points of view: load balancing, accuracy and complexity. Finally, Section V concludes this work.

II. Context

In this section, we introduce concepts needed throughout the rest of the paper.

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 formal definitions are as follows: 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 $\forall$ $s_i \in kNN(r, S)$, $\forall$ $s_j \in S - kNN(r, S)$, $d(r, s_i) \leq d(r, s_j)$.

It is easy to extend the previous definition to a set of query points.

Definition 2: The **kNN join** of two datasets R and S, kNN($R \ltimes S$) is:

$$\text{kNN}(R \ltimes S)=\{(r,\text{kNN}(r,S)),\ \forall\ r \in R\}$$

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 exact kNN, this definition can be extended to an approximate kNN join.

Definition 4: The $(1 + \epsilon)$**-approximate kNN join** of two datasets R and S, AkNN($R \ltimes S$) is:

$$\text{AkNN}(R \ltimes S)=\{(r,\text{AkNN}(r,S)),\ \forall\ r \in R\}$$

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 data sets. 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 using efficient data structures. For example, a one-dimension index structure, the B^+-Tree, is used in [8] to index distances; [9] adopts a multipage overlapping index structure R-Tree, whose minimum bound is a rectangle (MBR); [10] proposes to use a balanced and dynamic M-Tree to organize the dataset; [17] introduces a sphere-tree with a sphere-shaped minimum bound to reduce the number of areas to be searched; [18] presents a multidimensional quad-tree in order to be able to handle large amount of data; and [12] develops a kd-tree which is a clipping partition method to separate the search space.

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) [19] and Space Filling Curve [20]. In these cases, the low-dimension data contain much less information than the high-dimension ones because in most cases, the low-dimension data cannot totally fill the high-dimension space. To keep as much information as possible, the trick is to project the high-dimension data multiple times from different perspectives, producing multiple new datasets. For instance, LSH usually provides many space hashing functions. The space filling curve method also usually needs several shifts of data, to reduce the possibilities of errors in the data locality when projecting.

But with the increasing amount of data, these methods can still not handle kNN computation on a single machine efficiently. Experiments in [21] 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.

Recent papers have focused on providing efficient distributed implementations. Some of them use ad hoc protocols based on well-known distributed architectures [22], [23]. But most of them use the MapReduce model as it is naturally adapted for distributed computation, like in [24]–[26]. In this paper, we focus on the kNN computing systems based on MapReduce, because such systems can scale with the predictable data growth, and also because the framework on which they are based is widespread.

B. 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 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 overall workflow of a MapReduce computation is shown in Figure 1.

This decomposition enables data parallelism as the only dependency between the output of the maps and input of reduces. In practice, the map and reduce functions can be executed on many machines, leading to a naturally distributed computation. The most famous implementation of this model is the Hadoop framework [14] which provides a distributed platform for executing MapReduce jobs.

III. Workflow

In this section, we review the methods that have been used to perform kNN join on MapReduce. Overall, they all share the same workflow, comprised of three stages: (i) data preprocessing (ii) data partitioning and organization and (iii) computation of kNN. We will focus on the following works: the initial (basic, not optimized) idea, which we call H-BkNNJ , and its improvements H-BNLJ and H-BRJ in [26], as well as more advanced solutions such as H-zkNNJ in [26], RankReduce in [24] and PGBJ in [25].

A. Data Preprocessing

The preprocessing stage aims either at processing the initial dataset to reduce its complexity or produces extra information

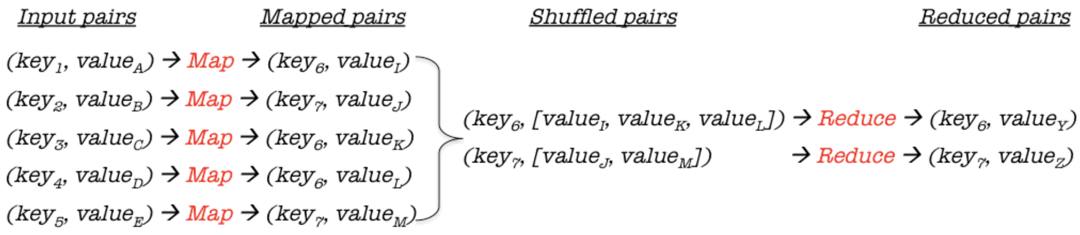
Input pairs
Mapped pairs
Shuffled pairs
Reduced pairs
(key1, valueA) → Map → (key6, valueI)
(key2, valueB) → Map → (key7, valueJ)
(key3, valueC) → Map → (key6, valueK)
(key4, valueD) → Map → (key6, valueL)
(key5, valueE) → Map → (key7, valueM)
(key6, [valueI, valueK, valueL]) → Reduce → (key6, valueY)
(key7, [valueJ, valueM]) → Reduce → (key7, valueZ)

Fig. 1. MapReduce workflow

about the data. It is an optional step, as some algorithms can directly work on the initial data, as H-BkNNJ, H-BNLJ and H-BRJ [26].

For high-dimensional data, the first approach wants to project data onto a low-dimensional one. One way to reduce the complexity and size of the dataset is to use space-filling curves as shown in [20]. The solution consists in mapping high-dimensional data to low-dimensional one while maintaining the locality relationship between each object with high probability (two elements that are close in a high-dimensional space remain close in the reduced dimensional space). In [26], the authors compute the z-value of all data from R and S. The z-value of a data is a one dimensional value that is calculated by interleaving the binary representation of the coordinates from MSB (Most Significant Bit) to LSB (Least Significant Bit). However, the ability of z-value to maintain the relative locality between objects in space is not good enough, especially when the dimension is high. Therefore, in practice, several random shifts of the data are generated, and additional z-values are computed, which aims to improve the accuracy at the cost of computation and space. Another way to reduce the dimension of data is LSH. In RankReduce [24], high-dimensional data are projected onto low-dimensional ones using *Locality Sensitive Hashing (LSH)*. The idea is to use a group of locality preserving hash functions to map close points from the original data to the same hash value with high probability. The overall performance of LSH depends on parameter tuning [27] which depends on the original dataset. Although it is likely for related objects to have the same hash values, in general, one single hash function cannot guarantee a satisfying accuracy. Often, a group of hash functions is required in order to generate multiple hash tables to reduce the number of collisions for distant objects.

In fact, as the quality of the projection is data dependent, both solutions in [26] and [24] duplicate the initial dataset and use different parameters for the projection to end up with several projected datasets. The purpose of this duplication is to alleviate information loss from the initial dataset: having multiple projected datasets enable to isolate potential errors. Overall, those solutions are willing to pay the same computation several times on different projected datasets in trade for the complexity of data.

Another approach consists in selecting leaders in the dataset which will drive the subsequent computation. In PGBJ [25], the preprocessing phase tries to identify pivots points (points that correspond more or less to the barycenter of a cluster of points) in the initial dataset which will lead a partition of the data. Thus, it is a primary selection before the partitioning stage. In paper [25], three strategies to select the pivots are described. The ”Random Selection” strategy generates a set of samples, and calculates the pairwise distance for every points in the samples, and then sums all the distances together. Then, the sample with the biggest summation of distances will be chosen as set of pivots. This strategy provides good results as long as the sample sets are big enough, to maximize the chance to select points in different clusters. The ”Farthest Selection” strategy randomly chooses the first pivot. Then, the farthest point to the first pivot will be chosen as the second pivot, and so on until having the desired number of pivots. This strategy ensures that the distance between each selected points is as large as possible, but it is heavier to process than the previous one, as it requires the computing of a lot of distances. Finally, the ”K-Means Selection” applies the traditional k-means method on a data sample to update the centroid of a cell as the new pivot each step, until the pivots do not change. With this strategy, the pivots are ensured to be in the middle of a cluster of points, but it is the heaviest strategy as it needs to converge towards the optimal solution. As we will see in the next section, the quality of pivots has an important impact on the quality of the partitioning.

To sum up this part, preprocessing methods are based on two strategies: projection or pivots selection. The first method tends to lose some information about the data but greatly reduces the complexity of the subsequent steps. On the contrary, the second method is more tedious to perform but preserves data, as it just a selection and not a transformation.

B. Data Partitioning and Organization

The first attempt to compute kNN efficiently in shared-memory focused on particular data organizations, such that the neighbor set can be pruned and neighbor sorting is performed faster. In the most popular methods, data are usually indexed using a tree structure like B^+-Tree [8] or R-Tree [9]. But as we target big data, shared-memory centric solutions cannot be easily applied to a shared-nothing platform as MapReduce. Instead, data need to be separated into several sets, called partitions, such that, ideally, each partition is independent from others. It is nonetheless possible to use efficient data structures to improve local searching in a partition.

As in any MapReduce computation, the data partition strat-

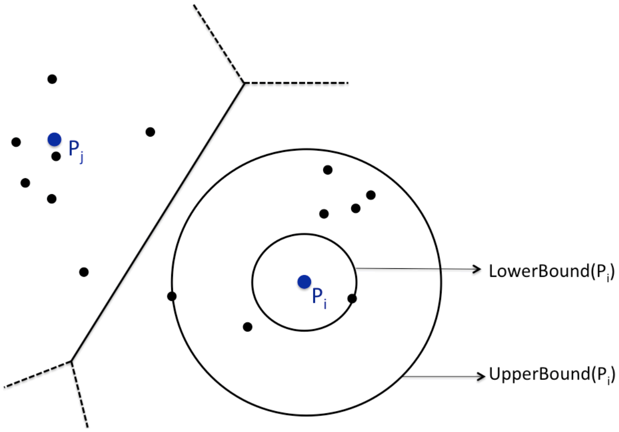

Fig. 2. Voronoi based partitioning for pivots P_i and P_j

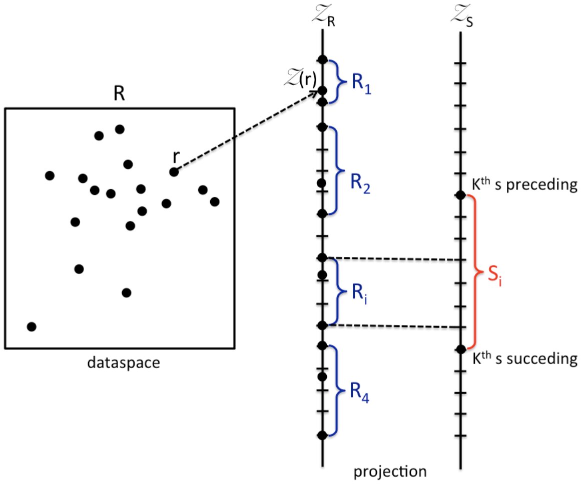

Fig. 3. z-value based partitioning

egy will strongly impact CPU, network communication and disk usage, which in turn will impact the overall processing time [28]. The key to improve the performance is to preserve spatial locality of objects when decomposing data for tasks [29]. 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 as an element in a partition of R will have its nearest neighbors in only one partition of S. More precisely, what we want is:

For every partition R_i ($\cup_i R_i = R$), find a corresponding partition S_j ($\cup_j S_j = S$), where

$$kNN(R_i \ltimes S) = kNN(R_i \ltimes S_j), \text{ and,}$$
$$kNN(R \ltimes S) = \bigcup kNN(R_i \ltimes S_j)$$

which means that, not only it is possible to compute kNN for each element of R_i in a single S_j, but also the concatenation of the results for all R_i is equal to the global kNN join.

In this section, we present two partition methods that enable to separate the data set into shared-nothing subset while preserving locality information.

1) Distance Based Partitioning: The first partitioning method 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. Because this method relies on the distance metric, it is naturally used to solve neighborhood problems. More formally, the definition of a Voronoi cell is as follow:

Definition 5: 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: ∀ i ≠ j, $VC(p_i) = \{p \| d(p, p_i) \leq d(p, p_j)\}$.

Paper [25] gives a method to partition the dataset R and S using Voronoi Diagrams. After having identified the pivots in R (as seen in the preprocessing section - III-A), they compute the distance between every point and the pivots, and put the considered point into the cell of the closest pivot. This will naturally give a partitioning of data. Afterwards, they also calculate, for each cell of R, the upper (resp. lower) bound of the partition which corresponds to the sphere determined by the furthest (resp. closest) point (in S) from the pivot in the cell of R. Such bounds enable to quickly find the correct partition S_j for a given partition R_i. This data partitioning principle is shown in Figure 2 where two pivots P_i and P_j have been chosen from the dataset.

The main problem of this method is that it requires computing the distance of all elements to the pivots. Also, the distribution of the input data might not be known in advance. Hence, the pivots will have to be recomputed if the data changes, which limits dynamicity. Also, there is no guarantee that all cells will 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.

2) Size Based Partitioning: Another type of partitioning method aims at dividing the data into some equal size partitions while preserving their locality information. [26] proposes a partitioning strategy based on z-value described in the previous section. In order to make every partition of R have a similar number of objects, a sampling is first performed. They claim that the n quantiles (n is equal to the number of partitions) of the sampling data is an unbiased estimation of the boundary points of each partition, with a standard deviation $\leq \epsilon |R|$ $(\epsilon > 0)$. After having partitioned R this way, the same sampling strategy is applied to S. For each of the boundary points of all R_i, the corresponding k^{th} nearest neighbor is identified in S. These points are then used as boundary points of the corresponding S_i. As a consequence, the partitions of S are overlapping such that for any given point in R, all the k nearest neighbors can always be found in a single partition of S. An example of z-value based partition is given in Figure 3. This method is likely to produce a substantially equivalent number of objects in each partition.

Another similar size based partitioning method can be applied for datasets that are preprocessed with *Locality Sensitive Hashing*, as illustrated in Figure 4. With this method, the elements of R that are projected to $LSH(R)$, are divided into quasi-equal size partitions using a sampling technique. The

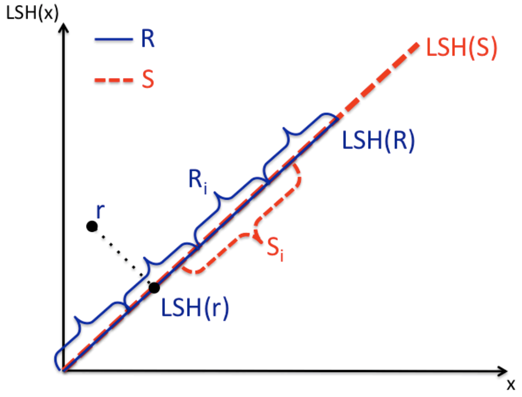

Fig. 4. LSH based partitioning

same technique can be applied to S and $LSH(S)$.

The strategy of partitioning will impact directly the number of tasks and the amount of computation. Distance based method aims at dividing the close objects together by preselecting some pivots. Size based method wants to separate objects into equal size zones in which the points are ordered. Regarding the implementation, [25] uses a MapReduce job to perform the partitioning. In [26], both data preprocessing and data partitioning are completed in one MapReduce job.

C. Computation

The general idea to compute a kNN, is to (i) calculate the distance between r_i and s_j for all i and all j, and (ii) sort these distances in ascending order to find out the first k results. In MapReduce, the idea is the same except that what is given to a task must be independent to ensure correctness without duplicating the whole dataset. The number of MapReduce jobs used for computing and sorting has an impact on the global performance, given the complexity of the computation performed by each task and the amount of data to exchange between them. The preprocessing and partitioning steps can also affect the number of tasks that are further needed by each MapReduce job. In this section, we review the different strategies used to compute and sort distances efficiently using MapReduce. These different strategies can be divided into two categories: the ones using one round of MapReduce job and the ones using two rounds of MapReduce jobs. Then, those categories can 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. We detail all these strategies in the following.

1) One Round of MapReduce Job: **Without preprocessing and partitioning** The basic idea (H-BkNNJ) to compute a kNN with MapReduce is to have every Map task process a pair of R_i and S_j, and perform a block nested loop on them to calculate the distance between $r_i \in R_i$ and $s_j \in S_j$, $\forall$ i and j. Note that, without any smart partitioning strategy, every possible blocks of one partition R_i from R and S_j from S should be calculated, leading to n^2 tasks totally where n is the number of partitions of R and S. The output of the Map task is in the form of $(id(r_i), list\,(id(s_j), d\,(r_i, s_j)))$. The identifier of r_i, named $id(r_i)$, is used as a key and the associated value is a list containing the identifier of s_j and the computed distance between r_i and s_j. A reduce task then processes all computed distances for a given r_i, and sorts them in ascending order to output the top k results.

With preprocessing and partitioning To reduce the number of tasks used to calculate the distances, in other words, to reduce the number of pairs formed by R_i and S_j, PGBJ [25] uses a preprocessing and Distance Based partitioning strategy, which ensures that each partition R_i has only one corresponding partition S_i needed to be searched, which reduces the number of tasks to n. Once the blocks of R_i and S_i identified, they perform the same calculation as in the H-BkNNJ technique. Because of their data organization, they are able to reduce the number of tasks, greatly improving the performance.

In RankReduce [24][1], the authors first preprocess and partition data into buckets using LSH. Their Map function computes the distance of all points in the partition, and then sort the local distances to output the local k nearest neighbors for each point in the partition, together with their distance in form of $(id(r_i), list\,(id(s_j), d\,(r_i, s_j)))$. These key-value pairs are then pulled by Reducers where they are sorted and issued as final results.

Overall, the main limitation of these approaches is that the number of values to be sorted in the reduce phase can be extremely large, up to $|S|$, if the pre-processing and partitioning have not significantly reduced the set of candidates. This greatly limits the applicability of such approach.

2) Two Rounds of MapReduce Jobs: **Without preprocessing and partitioning** To overcome the previously described limitation, multiple successive MapReduce jobs are required. The idea is to have the first job output the 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 perform the merging and sorting of all local top k values (instead of all values), producing the final global top k. Such approach is used in H-BNLJ [26] and greatly improves sorting time.

With preprocessing and partitioning A last possibility is based on the z-value preprocessing. In H-zkNNJ [26] the authors propose to generate some shifted copies of R and S (to improve the accuracy), and to determine the bound of the partitions of R_i and the corresponding S_i in a preprocessing MapReduce job. So here, the preprocessing and partitioning step is completely integrated in MapReduce. Then, the first MapReduce round of computation takes the partitions R_i and S_i previously determined, and computes the candidate neighbor set, named $C_i\,(r)$, for $\forall r_i \in R_i$, which contains k local nearest objects immediately before r_i and k objects after r_i ($2k$ objects totally). The second MapReduce round decides the exact result for $\forall r_i \in R$ from the candidate set $kNN\,(r, C_i\,(r))$. So in total, three MapReduce jobs are

[1] Although RankReduce only compute kNN for a single query, it is easy to extend it to a full kNN join

launched, and among them, two are actually devoted to the kNN 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.

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, and we review two main techniques. The first preprocessing technique, based on projecting the data to a one-dimension space, facilitates the next steps, because it is easier to manipulate data in one dimension rather than in high dimensions. However, data locality might not be preserved through this technique, and this is why, for a given set of initial data, several projections are performed with different seeds, to be able to spot errors. Consequently, the resulting dataset contains more data than the original one[2]. The second preprocessing technique is not based on transforming the data but rather on selecting the data that are significant in the dataset, like the ones being in the middle of a cluster of data. To find those significant data, a sampling must be done on the initial dataset. This technique has no impact on the data itself.

The second step of the workflow focuses on data partitioning and organization in the partitions. The distance based partitioning technique divides the data according to their distance with given reference points. This technique is straightforward although it does not guarantee finding the kNN in a single partition, because there is no bound on the number of points in a given partition. The size based technique can address this problem by constructing partitions of a given size. However, there is no guarantee that all the kNN will be found in a given partition unless sophisticated schemes are used when creating the partitions. During the partitioning step, data can also be organized using efficient data structures that offer good properties for searching and sorting.

The last step of the workflow is to actually compute the kNN. This step can be done in one or two MapReduce jobs. Choosing one way or the other essentially depends on the number of distances that need to be sorted.

Figure 5 summarizes the workflow we have gone through in this section and the techniques that are associated with each step. As those steps are more or less independent from each other, it is possible to focus on a particular step and choose the technique that best fits the use case, without impacting the next step. Hence it is possible to create a new kNN workflow by carefully selecting the algorithm at each step. For example, one can choose to transform the data with LSH or z-value in the pre-processing stage, then split them into blocks with Size Based Partitioning method, and use one or two rounds of MapReduce jobs for computation and sort.

IV. Analysis

In this section, we analyze existing works from another point of view than the workflow they follow. We first analyze load balancing to figure out the impact of the partitioning strategy. We then point out solutions that produce an exact result from solutions that produce an approximate result, in order to complete faster. We also compare the expected complexity of the systems. Finally, we consider the problem of changes in the dataset (either in R or S) and the challenges it raises.

A. Load Balance

In a MapReduce job, the completion time of the Map phase and the Reduce phase depends on the longest-running task. This makes load balancing particularly important. In this sense, the partitioning strategy should ensure that the processing time of each task is roughly the same, so as to achieve the optimal overall performance. In our context, each task is used to calculate the pairwise distance of $< r_i, s_i >$ in the partition $< R_i, S_i >$. Thus, the number of distances that need to be calculated is $|R_i| \times |S_i|$. If all tasks have to compute the same number of distances, they will have a roughly equivalent duration. Hence, the optimal partition strategy should make:

$$\forall \text{ i} \neq \text{j}, |R_i| \times |S_i| = |R_j| \times |S_j|.$$

This is hard to ensure in practice. It is however possible to obtain a sub-optimal solution in the following situation. Since S_i is the set of possible nearest neighbors for the elements in R_i, then:

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

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 (1) $|R_i| = |R_j|$ or (2) $|S_i| = |S_j|$.

In [26], the authors give a short proof which shows that the worst-case complexity for (1) is equal to:

$$O\left(|R_i| \times log\,|S_i|\right) = O\left(\frac{|R|}{n} \times log\,|S|\right) \tag{1}$$

and for choice (2), the worst-case complexity is equal to:

$$O\left(|R_i| \times log\,|S_i|\right) = O\left(|R| \times log\frac{|S|}{n}\right) \tag{2}$$

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

B. Accuracy

Among the systems we have introduced, one crucial property that we have not discussed yet is whether the system

[2] The new data set is actually the union of the data that arise from the different projections

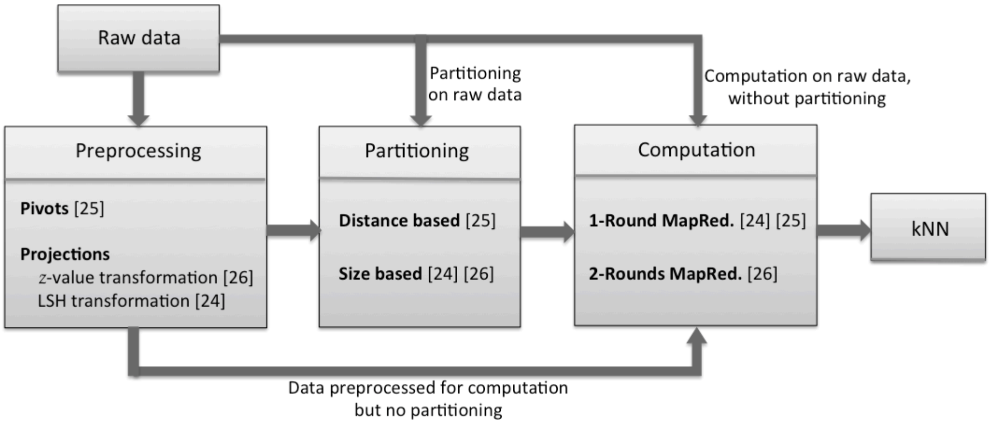

Fig. 5. Usual workflow of a kNN computation using MapReduce

produces a reliable answer or not. In order to quickly get answers, some systems prefer to loosen the accuracy of the kNN computation. That is, in an approximate kNN computation, the results are the same as in an exact kNN computation with a high probability. Nevertheless, results can be sub-optimal in some cases and even incorrect with a low probability. This can be acceptable in situations where responsiveness is crucial, for example when there are user interactions. In some other cases, only an exact kNN computation is appropriate. Hence it is an important criteria, depending on the use cases.

Usually, the lack of accuracy is the direct consequence of techniques such as z-values and LSH used in the preprocessing step. In [26] (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. Experiments show that three shifts of the initial dataset (in dimension 15) are sufficient to achieve a good approximation (less than 10% of errors shown in the experiences), while controlling the computation time. A similar result can be obtained with LSH by increasing the number of hash functions used. However, when using space filling curves, the distance from the optimal solution (the true kNN) is often bounded, which is not the case for LSH, as explained in [30].

Although approximate, these solutions are indeed fully usable in practice and can provide a high level of confidence, depending on the quality of results required by the application. Nonetheless, error bounds are difficult nay impossible to compute, and mostly experiments enable to find the best parameters for a given approximate computation.

C. Complexity

Computational complexity is often used to describe the execution time of an algorithm. When computing kNN with MapReduce, additional factors strongly impact the execution time:

(1) **The number of MapReduce jobs:** starting a job (whether in Hadoop [31] or any other plateform) 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 network during the Shuffle phase. Moreover, scheduling a task also incurs an overhead.

(3) **The number of distances to compute and sort for each object r_i:** Sorting is a dominating operation, so the number of elements to be sorted also impacts computation time.

The basic method H-BkNNJ only uses one MapReduce job, and requires n^2 Map tasks to calculate the distances where n is the number of partitions. The complexity of sorting all distances for one r_i in R is $|S| \times log\,|S|$. Since S is usually a large dataset, this method quickly becomes impracticable.

To overcome this limitation, H-BNLJ or H-BRJ [26] uses 2 MapReduce jobs, again with n^2 Map tasks to compute the distances. However, using of a second job significantly reduces the complexity of sorting to $|n \cdot k| \times log\,|n \cdot k|$, where n is the number of partitions and k is the number of nearest neighbors queried.

PGBJ [25] performs a pre-processing phase followed by 2 MapReduce jobs. This method also only uses n Map tasks to calculate the distances. Overall, the sorting complexity is reduced to $|S_i| \times log\,|S_i|$.

H-zkNNJ [26] also begins by a pre-processing phase and uses in total 3 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 neighbors set, into account. Since $C_i\,(r_i)$ contains only $2k$ neighbors for each r_i, the complexity is now reduced to $|2 \cdot k| \times log\,|2 \cdot k|$.

D. Dynamicity

So far, all the solutions we have studied did not take into account potential changes in the dataset: they are designed for static computation of kNN. However, in real applications, it is likely to have to recompute the kNN against an updated set R (new queries) or an updated set S (new database), or

Systems	Preprocessing	Partitioning	Total jobs	Accuracy	Complexity		Dynamicity
					Tasks	Sort	
H-BkNNJ (basic)	None	None	1	Exact	n^2	$\|S\| \times log\,\|S\|$	Static
H-BNLJ [26] (Zhang et al.)	None	Size based	2	Exact	n^2	$\|n \cdot k\| \times log\,\|n \cdot k\|$	Static
RankReduce [24] (Stupar et al.)	LSH	Size based	2	Approximate	n	$\|n \cdot k\| \times log\,\|n \cdot k\|$	Static
H-VDkNNJ [25] (Lu et al.)	Pivots selection	Distance based (Voronoi Diagram)	2	Exact	n	$\|S_i\| \times log\,\|S_i\|$	Static
H-zkNNJ [26] (Zhang et al.)	Z-curve	Size based	3	Approximate	n	$\|2 \cdot k\| \times log\,\|2 \cdot k\|$	Static

TABLE I
SUMMARY TABLE OF kNN COMPUTING SYSTEMS WITH MAPREDUCE

even in some cases with both updated sets. A fair number of works have been devoted to efficiently handling updates in the datasets R and S [11], [32], but only in a centralized context.

On the other hand, the MapReduce model is not naturally suited to dynamic data, as a reducer needs all the values of a given key to complete its computation. Hence data modification during a job execution is not supported. Some support for changing datasets between jobs [33], [34] has been proposed, but there still remains the challenge of efficiently implementing kNN with such constraints.

Indeed, we have seen that efficient solutions for computing kNN on MapReduce require a fine partitioning of the data. Consequently, to handle data updates efficiently, one should avoid repartitioning the dataset at each update as it is a heavy step. However, the partitioning step is often driven by the data itself. This is why it is important, when choosing a solution, to know if the whole partitioning is driven by the partitioning of the R dataset or of the S dataset. For example, if the partitioning is driven by the partitioning of R as in [25], [26], then having new queries will force to retrigger the whole partitioning process (and potentially the preprocessing of data too) from the beginning. An application that frequently receives new data will probably not benefit from this technique. On the other hand, if the updates are located in the reference set S, the partitioning can be kept unchanged for several successive updates.

To the best of our knowledge, up to now, there is no dominating solutions for processing continuous kNN join on top of MapReduce.

E. 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 shrink 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. Finally, none of the systems really offers a way to handle data updates. This is due to the specific partitioning performed before computing. Indeed, an efficient partitioning is adapted to a particular dataset, and might not be adapted to another. Table I is a summary table of the systems we have examined and their main characteristics.

V. CONCLUSION

In this paper, we have studied the existing systems 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 the preprocessing of data, the partitioning and the actual computation. We have listed and explained the different algorithms which could be chosen for each step, and developed their pros and cons. In a second stage, we have further analyzed existing systems by reviewing their main properties, in terms of load balancing, accuracy of the computation, overall complexity, and management of data updates. 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.

REFERENCES

[1] D. Li, Q. Chen, and C.-K. Tang, "Motion-aware knn laplacian for video matting," in *International Conference on Computer Vision (ICCV), 2013*, 2013.

[2] H.-P. Kriegel and T. Seidl, "Approximation-based similarity search for 3-d surface segments," *Geoinformatica*, vol. 2, no. 2, pp. 113–147, Jun. 1998.

[3] X. Bai, R. Guerraoui, A.-M. Kermarrec, and V. Leroy, "Collaborative personalized top-k processing," *ACM Trans. Database Syst.*, vol. 36, no. 4, pp. 26:1–26:38, Dec. 2011.

[4] D. Rafiei and A. Mendelzon, "Similarity-based queries for time series data," *SIGMOD Rec.*, vol. 26, no. 2, pp. 13–25, Jun. 1997.

[5] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, ser. FODO '93. London, UK, UK: Springer-Verlag, 1993, pp. 69–84.

[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 (JCSSE), 2011 Eighth International Joint Conference on*, May 2011, pp. 128–131.

[7] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast nearest neighbor search in medical image databases," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 215–226.

[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 *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 426–435.

[11] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *GeoInformatica*, vol. 14, no. 1, pp. 55–82, 2010.

[12] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[14] "Hadoop." [Online]. Available: http://hadoop.apache.org/

[15] N. Bhatia and A. Vandana, "Survey of nearest neighbor techniques," *International Journal of Computer Science and Information Security*, vol. 8, no. 2, 2010.

[16] L. Jiang, Z. Cai, D. Wang, and S. Jiang, "Survey of improving k-nearest-neighbor for classification," in *Proceedings of the Fourth International Conference on Fuzzy Systems and Knowledge Discovery - Volume 01*, ser. FSKD '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 679–683. [Online]. Available: http://dx.doi.org/10.1109/FSKD.2007.552

[17] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *Geoinformatica*, vol. 14, no. 1, pp. 55–82, Jan. 2010.

[18] 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.

[19] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04. New York, NY, USA: ACM, 2004, pp. 253–262.

[20] 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.

[21] 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*, vol. 7, no. 8, Aug.

[22] D. Novak and P. Zezula, "M-chord: A scalable distributed similarity search structure," in *Proceedings of the 1st International Conference on Scalable Information Systems*, ser. InfoScale '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1146847.1146866

[23] P. Haghani, S. Michel, and K. Aberer, "Lsh at large distributed knn search in high dimensions ."

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

[25] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, Jun. 2012.

[26] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 38–49.

[27] M. Bawa, T. Condie, and P. Ganesan, "Lsh forest: Self-tuning indexes for similarity search," in *Proceedings of the 14th International Conference on World Wide Web*, ser. WWW '05. New York, NY, USA: ACM, 2005, pp. 651–660.

[28] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin, "A Hadoop MapReduce Performance Prediction Method," in *2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Zhangjiajie, Chine, Nov. 2013, pp. 820–825.

[29] X. Zhou, D. J. Abel, and D. Truffet, "Data partitioning for parallel spatial join processing," *Geoinformatica*, vol. 2, no. 2, pp. 175–204, Jun. 1998.

[30] S. Liao, M. A. Lopez, and S. T. Leutenegger, "High dimensional similarity search with space filling curves," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 615–622.

[31] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 472–483, Sep. 2010.

[32] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-nn queries on data streams," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 156–165.

[33] T.-T. Vu and F. Huet, "A Lightweight Continuous Jobs Mechanism for MapReduce Frameworks," in *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands*, Pays-Bas, Jun. 2013, pp. 269–276.

[34] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920881