A Survey on Approaches for Computing $k$ Nearest Neighbor on Map Reduce

Tejal D. Kateore$^1$, Prof. Suhasini Itkar$^2$

$^1,2$ Computer Department, P. E. S. Modern College of Engineering, Shivajinagar, Pune-05

Abstract- The $k$NN($k$ Nearest Neighbor) is frequently used in classification methods. The $k$NN operation is used to discover the joins between two data points $S$ and $R$. However, as the volume and the dimension of data increase, the performance of $k$NN degrades. Only distributed approaches can perform better for such costly operation. Here we focus on studying the different efficient solutions using the MapReduce programming model, as it is suitable for distributed large scale data processing. The different existing approaches such as H-BkNNJ, H-BNJ, PGBJ, H-zkNNJ, RankReduce, for computing $k$NN on MapReduce is done. Data pre-processing, data partitioning and computation are the three common steps for $k$NN computation on MapReduce. Then analysis is done for each step from load balancing, accuracy and complexity aspects.

General Terms- $k$ Nearest Neighbor, MapReduce, Performance Evaluation.

Keywords- Partition Group Block Join(PGBJ), Hadoop Block Nested Loop $k$NN (H-BNLJ), RankReduce, Hadoop $z$ value (H-zkNNJ).

I. INTRODUCTION

Given $R$ is a point and $S$ is set of reference points, a $k$ nearest neighbor join is an operation which for each point in $R$, discovers the $k$ nearest neighbor in $S$. $k$ Nearest Neighbor is frequently used as a classification or clustering method in machine learning or data mining. 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 $k$NN query on the training set will be performed to estimate its class membership. This process can be considered as a $k$NN join of the testing set with the training set.

The basic idea to compute a $k$NN join is to perform a pairwise computation of distance for each element in $R$ and each element in $S$. The difficulties mainly lie in the following two aspect: (1) Data Volume (2) Data Dimensionality. A lot of work has been dedicated to reduce the in-memory computational complexity[1]. These works mainly focus on two points: (1) Use indexes to decrease the number of distances need to be calculated. These indexes can hardly be scaled on high dimension data. (2) Use projections to reduce the dimensionality of data. But the maintenance of the accuracy becomes another issue. Despite these efforts, there are still significant limitations to process $k$NN on a centralized machine when the amount of data increases.

Only distributed and parallel solutions are proved to be powerful, for large dataset. MapReduce is a flexible and scalable parallel and distributed programming paradigm which is specially designed for data-intensive processing. Writing an efficient $k$NN in MapReduce is also challenging for many reasons. The goal of this paper is to survey existing methods of $k$NN in MapReduce, and to compare their performance. In [8], the authors only focus on centralized solutions to optimize $k$NN computation whereas we target distributed solutions. In [2], the survey is also oriented towards centralized techniques
and is solely based on a theoretical performance analysis. An analysis which outlines the influence of various parameters on the performance of each algorithm. The paper is organized into four sections: Section II gives brief review of the approaches for $k$NN computation using map reduce. Section III describes the performance parameters considered to compare these approaches and finally, Section IV summarizes and presents the conclusions.

II. RELATED WORK

$k$NN is based on a distance function that measures the difference or similarity between two instances. $k$NN using centralized approach was not able to perform for large inputs. So a new approach to execute it parallelly was developed. There are various existing solutions to perform the $k$NN operation in the context of MapReduce are given. All solutions follow three main steps to compute $k$NN over MapReduce, (1) preprocessing of data, (2) partitioning and (3) actual computation.

H-BNLJ[1] actually cannot guarantee load balancing, because of the random method it uses to split data. 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 $2n$ 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. Some reducers will have more elements to process than others, slowing the computation. Overall, the challenge with this algorithm is to find the optimal number of partitions for a given dataset.

H-z$k$NNJ [1], [3], which use size based partitioning strategies, have a very good load balance, with a very small deviation of the completion time of each task. In H-z$k$NNJ, the $z$-value transformation leads to information loss. The recall of this algorithm is influenced by the nature, the dimension and the size of the input data. More specifically, this algorithm becomes biased if the distance between initial data is very scattered, and the more input and $M$, the number of hash functions in each family. Since they are dependent on the dataset, experiments are needed to precisely tune them. In, the authors suggest this can be achieved with a sample dataset and a theoretical model. The first important metric to consider is the number of candidates available in each bucket. Indeed, with some poorly chosen parameter values, it is possible to have less than $k$ elements in each bucket, making it impossible to have enough elements at the end of the computation.

RankReduce [1],[4], with the addition of a third job, can have the best performance of all, provided that it is started with the optimal parameters. The most important ones are $W$, the size of each bucket, $L$, the number of hash families. Increasing the number of families $L$ greatly improves both the precision and recall. However, increasing $M$, the number of hash functions, decreases the number of collisions, reducing execution time but also the recall and precision. Overall, finding the optimal parameters for the Locality Sensitive Hashing part is complex and has to be done for every dataset.

For PGBJ, [1], [5] Greedy grouping gives a better load balance than Geo grouping, at the cost of an increased duration of the grouping phase. By exploiting partitioning method, the proposed approach is able to divide the input datasets into groups and we can answer the $k$ nearest neighbor join by only checking object pairs within each group. Several pruning rules are developed to reduce the shuffling cost as well as the computation cost. Extensive experiments performed on both real and synthetic datasets demonstrate that the proposed methods are efficient, robust and scalable. The communication overhead of PGBJ is very sensitive to the choice of pivots. The data are another important aspect affecting the performance of the algorithms. As expected, all the algorithms’ performance decreases as the dimension...
of data increases. However, what exceeded the prediction of the theoretical analysis is that the dimension is really a curse for PGBJ[8]. PGBJ has the lowest disk space requirement.

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 input size to use is defined by input split size. This algorithm is clearly not suitable for large datasets but because of its simplicity, it can, nonetheless, be used when the amount of data is small.

### III. PERFORMANCE PARAMETERS

The performance of the techniques is measured in following parameters[1].

- **Recall**: The recall measures the ability of an algorithm to find the correct $k$NN.
- **Precision**: The fraction of correct $k$NN in the final result set.

By definition, the following properties holds: (1) recall $\leq$ precision because all the tested algorithms return up to $k$ elements.

(2) if an approximate algorithms outputs $k$ elements, then recall = precision.

### IV. CONCLUSION

In this paper, we have studied the existing systems to perform the $k$NN operation in the context of MapReduce. We have first approached this problem from a work-flow point of view. We have pointed out that all solutions follow three main steps to compute $k$NN over MapReduce, namely the pre-processing of data, the partitioning and the actual computation. We have listed and explained the different algorithms which could be chosen for each step, and their pros and cons could be developed. In a second stage, we have further analyzed existing systems by reviewing their main properties, in terms of load balancing, accuracy of the computation, and overall complexity. Above all, this paper can be seen as a guideline to help selecting the most appropriate method to perform the $k$NN join operation on MapReduce for a particular use case.

### REFERENCES