Comparing MapReduce-Based k-NN Similarity Joins On Hadoop For High-dimensional Data

Přemysl Čech[1], Jakub Maroušek[1], Jakub Lokoč[1], Yasin N. Silva[2], and Jeremy Starks[2]

[1] SIRET research group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University in Prague
{lokoc,cech}@ksi.mff.cuni.cz,
marousej@artax.karlin.mff.cuni.cz
[2] Arizona State University
{ysilva,Jeremy.Starks}@asu.edu

Abstract. Similarity joins represent a useful operator for data mining, data analysis and data exploration applications. With the exponential growth of data to be analyzed, distributed approaches like MapReduce are required. So far, the state-of-the-art similarity join approaches based on MapReduce mainly focused on the processing of low-dimensional vector data. In this paper, we revisit and investigate the performance of different MapReduce-based approximate k-NN similarity join approaches on Apache Hadoop for large volumes of high-dimensional vector data.

Keywords: Hadoop, MapReduce, k-NN, Approximate similarity join, HTTPS data

1 Introduction

The k-NN similarity joins serve as a powerful tool in many domains. In the data mining and machine learning context, k-NN joins can be employed as a preprocessing step for classification or cluster analysis. In data exploration and information retrieval, similarity joins provide a similarity graph with the most relevant entities for each object in the database. Their applications can be found for example in the image and video retrieval domain [6, 7], and in network communication analysis and malware detection frameworks [2, 11]. Because data volumes are often too large to be processed on a single machine (especially for high-dimensional data), we study the use of the distributed MapReduce environment [5] on Hadoop[3]. Hadoop MapReduce is a widely adopted technology and considered an efficient and scalable solution for distributed big data processing.

Related papers [15, 16, 9] have deeply analyzed advantages, disadvantages and bottlenecks of distributed MapReduce systems Hadoop and Spark[4] [22]. In this paper, we study similarity join algorithms that were designed and implemented

[3] http://hadoop.apache.org/
[4] http://spark.apache.org/

on Apache Hadoop. The comparison considers methods employing data organization/replication strategies initialized randomly as they enable convenient application and usage on different domains. Although several studies tackling similarity joins have been previously published for Hadoop [19, 25], these algorithms focused mainly on data with lower dimensionality. The need of effective and efficient high-dimensional-data k-NN similarity joins led us to revise available MapReduce algorithms and integrate further adaptations. In the paper, we study three different approaches which offer diverse ways of approximate query processing with a promising trade-off between error and computation time (when compared to exact k-NN similarity joins).

The main contributions of this paper are revisions and adaptations of all algorithms for high-dimensional data and thorough experiments. Particularly, we report interesting findings for high-dimensional data that were not previously identified. We also discuss implementation difficulties and other modifications and compare all presented algorithms according to multiple testing scenarios and demonstrate scalability, competitiveness and suitability of all the solutions for high-dimensional real data (200, 512, 1000 dimensions).

The paper is structured in the following order. In Section 2, all essential definitions are presented. Section 3 summarizes all investigated k-NN similarity join algorithms with revisions. In Section 4, we examine the presented approaches in multiples experimental evaluations and discuss the results, and finally, in Section 5, we conclude the paper.

2 Preliminaries

In this section, we present fundamental concepts and basic definitions related to approximate k-NN similarity joins. All the definitions use the standard notations [19, 23].

2.1 Similarity model and k-NN joins

In this paper we address the efficiency of k-NN similarity joins of objects o_i modeled by high-dimensional vectors $v_{o_i} \in \mathbb{R}^n$. In the following text, a shorter notation v_i will be used instead of v_{o_i}. In connection with a metric distance function $\delta : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_0^+$, the tuple $M = (\mathbb{R}^n, \delta)$ forms a metric space that serves as a similarity model for retrieval (low distance means high similarity and vice versa)[5].

Let us suppose two sets of objects in a metric space M: database (train) objects $S \subseteq \mathbb{R}^n$ and query (test) objects $R \subseteq \mathbb{R}^n$. The similarity join task is to find the k nearest neighbors for each query object $q \in R$ from the set S employing a metric function δ. Usually, the Euclidean (L_2) metric is employed. Formally:

$$kNN(q, S) = \{X \subset S; |X| = k \wedge \forall x_i \in X, \forall y \in S - X : \delta(q, x_i) \leq \delta(q, y)\}$$

[5] Note that the effectiveness of the distance function and feature extraction mapping from o_i to v_i is the subject of similarity modeling.

and the k-NN similarity join is defined as:

$$R \ltimes S = \{(q, s)|q \in R, s \in kNN(q, S)\}.$$

Because of the high computational complexity of similarity joins, we focus on approximations of joins which can significantly reduce computation costs while keeping reasonable precision. Formally an approximate k-NN query for an object $q \in R$ is labeled as $kNN_a(q)$ and defined as ϵ-approximation of exact k-NN:

$$kNN_a(q, S) = \{X \subset S; |X| = k \wedge$$
$$\max_{x_i \in kNN(q,S)} \delta(q, x_i) \leq \max_{x_i \in X} \delta(q, x_i) \leq \epsilon \cdot \max_{x_i \in kNN(q,S)} \delta(q, x_i)\}$$

where $\epsilon \geq 1$ is an approximation constant. The corresponding approximate k-NN similarity join is defined as:

$$R \ltimes_a S = \{(q, s)|q \in R, s \in kNN_a(q, S)\},$$

For high-dimensional vector representations, all the pairwise distances between dataset vectors tend to be similar and high with respect to the maximal distance (the effect of high intrinsic dimensionality [23]). Hence, the ϵ constant for such datasets and given k would have to be very small to guarantee a meaningful precision with respect to exact search. At the same time, any filtering method implementing such ϵ guarantee would result in inefficient (i.e., too expensive) approximate kNN query processing. Therefore, in this work we do not consider such guarantees for the compared methods (theoretical limitations of the guarantees are out of the scope of this paper). In the experiments, we focus just on the error of the similarity join approximation. The error is measured as the mean of the approximation errors of particular k-NN queries. The k-NN query approximation precision (or recall with respect to the exact k-NN search) is defined as:

$$precision(k, q, S) = \frac{|kNN(q, S) \cap kNN_a(q, S)|}{k}$$

2.2 MapReduce environment

Since data volumes are significantly increasing every day, centralized solutions are often intractable for large data processing. Memory of a single computer is becoming insufficient, and CPUs do not provide enough power for query processing in reasonable time. Therefore, the need for effective distributed data processing is emerging.

In this paper, we have adopted the MapReduce [5] paradigm that is often used for parallel processing of big datasets. The algorithms described in Section 3 are implemented in the Hadoop MapReduce environment which consists of several components.

Datasets are stored in the Hadoop distributed file system (HDFS), which is designed to form a big virtual file space to contain data in one place. Data

files are physically stored on different data nodes across the cluster and are replicated in multiple copies (protection against a hardware failure or a data node disconnection). Name nodes manage access to data according to the distance from a request source to a data node (it finds the closest data node to a request).

In Hadoop, every program is composed of one or more MapReduce jobs. Each job consists of three main phases: a map phase, a shuffle phase and a reduce phase. In the map phase, data are loaded from the HDFS file system, split into fractions and sent to mappers where a fraction of data is parsed, transformed and prepared for further processing. The output of the map phase are `<key, value>` pairs. In the shuffle phase, all `<key, value>` pairs are grouped and sorted by the key attribute and all values for a specific key are sent to a target reducer. Ideally, each reducer receives the same (or similar) number of groups to equally balance a workload of the job. In the reduce phase all reducers process values for an obtained key (or multiple keys) and usually perform the main execution part of the whole job. Finally, all computed results from the reduce phase are written back to the HDFS.

3 Related k-NN similarity joins

In this paper we study a pivot-based approach for general metric spaces and two vector space approaches - space filling Z-curve and locality sensitive hashing.

3.1 Pivot-based approach

The original version of this approximate k-NN join algorithm [2] utilizes pivot space partitioning based on a set of preselected global pivots P_i. This approach was inspired by the Lu et al. work [12], which focused on exact similarity joins. The algorithm is composed of two main phases: the preprocessing phase and the actual k-NN join computation phase.

In the preprocessing phase, both sets of database and query objects (S and R) are distributed into Voronoi cells C_i using the Voronoi space partitioning algorithm according to the preselected pivots P_i (a cell C_i is determined by the pivot P_i). Next, all distances d_{j_i} from objects $o_j \in S \cup R$ to all pivots P_i ($d_{j_i} = d(o_j, P_i)$) have to be computed, and for every object o_j the nearest pivot P_n with the distance d_{j_n} is stored within the o_j data record. Also, global statistics are evaluated for every Voronoi cell C_i such as covering radius, number of objects o_j and total size of all objects o_j in the particular cell C_i. At the end of the preprocessing phase, the Voronoi cells C_i are grouped together into bigger groups G_l ($G_l = \cup_{i \in l} C_i$). Every group G_l should contain objects of a similar total size to properly balance further parallel k-NN join workload.

The second phase performs k-NN join of two sets S and R in a parallel MapReduce environment (one MapReduce job). Every computing unit (one reducer red_l) receives a subset $S_l \subset S$ of database objects and $R_l \subset R$ of query objects corresponding to a group G_l precomputed in the previous phase. Because not all nearest neighbors-valid candidates for query objects $q_l \in R_l$ may

be present in a group G_l (especially for query objects near G_l space boundaries), the replication heuristic is employed. Database objects $o_n \in S^n \in C^n$ ($S^n \subset S \wedge \forall o_n \in S^n, o_n \in C^n$) from the nearest Voronoi cells $C^n \subset C$ are replicated into the group G_l (reducer red_l). Specifically, every database object $o_l \in S_l$ from a cell C_l is replicated to all cells $C^n \subset C$ and corresponding groups $G^m \subset G$ where $|C^n| = ReplicationThreshold$ (constant determining number of replications), $G^m = \cup\, G_i$: $G_i \cap C^n \neq \emptyset$ and $\forall P_x \in C^n, \forall P_y \notin C^n : d(P_l, P_x) \leq d(P_l, P_y)$, $P_l \in C_l$. Additional details of this algorithm can be found in the original paper [2]. The output of a reducer red_l is a set of the k nearest neighbors for every query object $q_l \in R_l$.

An overview of the space partitioning and replication algorithm is depicted in Figure 1.a.

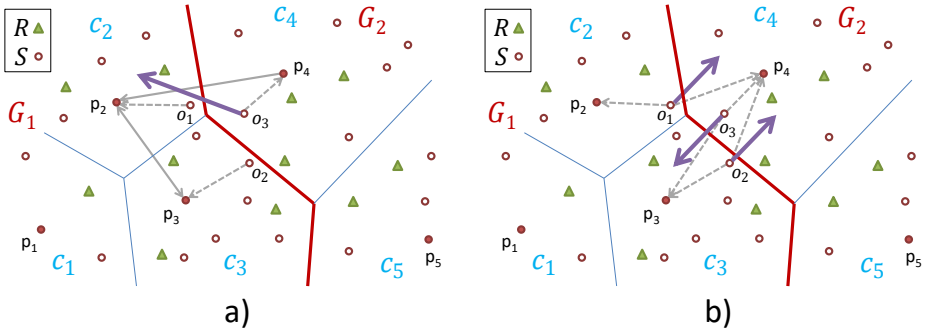

Fig. 1: An example of the Voronoi space partitioning and replication of database objects $o_n \in S$. The first part a) depicts the replication based on distances between pivots p_i. For the $ReplicationThreshold = 2$ only the object o_3 is replicated to the other group G_1, whereas o_1 and o_2 have the closest pivot to the corresponding pivot p_i (in the cell c_i) in the same group. In the b) scenario for the $MaxRecDepth = 2$ all three objects o_n near groups boundaries are replicated to the other group because the second closest pivot to the objects o_n lies in the other group.

Algorithm revision In this paper, we use a slightly modified version of the previously described algorithm. The main difference is the utilization of a repetitive (recursive) Voronoi partitioning inspired by indexing techniques in metric spaces such as M-Index [17]. Basically, every object o_j is identified by a pivot permutation [3] determined by a set of closest pivots instead of a single closest pivot. The modification influences mainly the preprocessing phase and also the database objects replication heuristic. The new algorithm is summarized in Figure 1.b.

We define a new parameter $MaxRecDepth$ which sets a threshold for the maximum depth of the Voronoi space partitioning. In the preprocessing phase, for every object o_j ($o_j \in S \cup R$) a set of n distances d_{j_i} ($d_{j_i} = d(o_j, P_i)$) is selected

and stored with the nearest pivots ${P_j}^n \subset P$ where $|{P_j}^n| = MaxRecDepth$ and $\forall x \in n, \forall y \notin n : d_{j_x} \leq d_{j_y}$.

The replication heuristic in the beginning of the second phase utilizes repetitive partitioning in a similar way to the original approach but the nearest cells C^n are not determined by the distance of corresponding pivots but precomputed in the preprocessing phase. Specifically, every database object $o_l \in S_l$ from a cell C_l is replicated to all cells $C^n \subset C$ and corresponding groups $G^m \subset G$ where $|C^n| = MaxRecDepth$, $G^m = \cup\, G_i$: $G_i \cap C^n \neq \emptyset$ and $\forall P_x \in C^n : P_x \in {P_l}^n$ (${P_l}^n$ are stored nearest pivots to the object o_l).

3.2 Space filling curve approach

Yao et al. [21] proposed the use of space-filling curves for approximate k-NN computation in relational databases. A space-filling curve is a bijection which maps an object in n-dimensional space to a one-dimensional value, preserving original objects locality with high probability. Of these functions, the authors chose z-order curve, whose value (referenced as *z-value*) can be computed easily by interleaving binary representation of coordinate values. When querying the database, the z-value of the query object is calculated and k database objects with nearest z-values are returned. Sorting the objects by their z-values in advance, the querying step can be done efficiently in logarithmic time.

While the z-curve preserves locality of points with high probability, some nearest neighbors might be omitted. To reach a more precise solution, c independent copies of the database are produced, each of them shifted by a random vector $v_i \in \mathbb{R}^n, i \in \{1 \dots c\}$; that means, for any object with coordinates vector p, the result value in copy C_i is equal to $p + v_i$. For database copy C_i, z-values of modified objects are computed and sorted in a list L_i.

When querying the database, the query object is shifted by each v_i as well, producing a vector of c z-values. For the i-th value, L_i is queried and the k nearest values are taken. Thus, $c \cdot k$ candidates are collected in total, their distance to the query object is computed and k nearest candidates are returned.

The centralized solution has been adapted for the MapReduce framework and was originally published in the paper by Zhang et al. [24]. Both the database and query objects are stored in HDFS. The random vectors and multiple copies approach is utilized as well. Moreover, to distribute the work among the nodes, the objects in each copy are split in n partitions, depending on their z-value. Inside each partition, we take all query objects and find k nearest database object candidates. Each partition is processed by a separate reducer. Using a suitable number of partitions and having data equally distributed, the portion of data for each reducer is small enough to be stored in a node memory.

Every database object belongs to exactly one partition. We must be ensured, however, that the partition contains all nearest neighbor candidates. That is, in each partition, a query object with maximum z-value needs k database objects with higher z-values copied over (if any exist). Analogically for a query object with minimum z-value. Since the intention is to distribute the objects equally, the best boundary points would be $\frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}$ quantiles of L_i. In

the MapReduce environment, however, this is expensive to compute: instead, objects are sampled and depending on their values, approximate quantiles are determined. The solution uses three MapReduce jobs. The first job, while initializing, produces random vectors and saves them. The mappers compute z-values for every object and shift, emitting tuples *(objectId, shiftId, zValue)* and writing them back to HDFS. In addition, with probability ϵ, the object is sampled and the corresponding z-values are sent to the reduce phase. For every shift, a reducer loads all sampled objects in the memory and, based on the distribution of z-values, determines the range values for partitions and writes them back to HDFS for the next step.

In the second job, a mapper takes a previously created tuple and emits a `<key, value>` pair for every partition that the object belongs to. A reducer accepts the data of one of the partitions and loads them in memory. Using the z-values, it finds the k nearest database objects for each query object. Transforming z-values of the objects, the coordinates are regained. Distances between a query object and each candidate are determined using the original metric function d. The algorithm emits tuples *(objectId, candidateId, distance)* and writes them back to HDFS.

For every query object, this approach produces $c \cdot k$ candidates. The purpose of the third job is to compare the distances. For each query object, its k closest candidates are considered as the output of the k-NN join.

Implementation revision The Java source code of the MapReduce solution was provided by its authors. We modified it and adjusted the data structures to fit our object representations. For the purposes of high-dimensional data computation, it was important to find a compact way how to represent and serialize z-values. In the original code, String objects were utilized; we edited the classes to use BigInteger objects instead.

We also altered the behavior of the reducers used in the second phase. The original source code caches the objects of a partition in a local (node) file system. This method is advantageous for large partition sizes. For our data, however, we found it better to load the objects directly to a node memory. This approach has the advantage of faster running times.

The original z-curve functions assume integer dimension values. Since our datasets contain float values, we transformed and rescaled all values to fit integer structures. Notice this does not change the nearest neighbors sets.

The algorithm itself has not been modified.

3.3 Locality sensitive hashing approach

Locality Sensitive Hashing (LSH) [8, 4] is another technique that has been used in the context of k-NN Similarity Join algorithms. Specifically, Stupar et al. proposed RankReduce [20], a MapReduce-based approximate algorithm to solve the k-NN Similarity Join problem using LSH. The key idea behind the RankReduce approach is to integrate hashing techniques to assign similar objects to similar

fragments in the distributed file system. This distribution enables an effective generation of candidate neighbors to identify the k nearest neighbors.

RankReduce is composed of a pre-processing step and a single MapReduce job. During the pre-processing step, an LSH index is built using a set of j hash functions of the form $h_{a,B}(v) = \lfloor (a \cdot v + B)/W \rfloor$. The evaluation of these functions on each input record v generates a result vector of length j. Each different value of this vector represents a bucket of data in the LSH index (containing all the associated records). W is a tuning parameter and affects the number of generated buckets. After this step, the algorithm applies the same set of hash functions to the query points and identifies the subset of buckets that will be used in the MapReduce job.

The Map function of the MapReduce job receives a subset of records $v_n \in S$ and the set of query points R, and computes the distance between the record and each query point. The Map stage keeps track of the local k-nearest neighbors of each query point q_i. The cleanup subroutine of the Map stage outputs the local k-nearest neighbors of each query point q_i. The Map output has the form $\langle (q_i, dist(q_i, v_n)), v_n \rangle$.

The Reduce function receives all the local k nearest neighbors identified in multiple Map nodes and selects the global k nearest neighbors.

Implementation revision Since we could not obtain the source code from the authors of the original paper, we implemented this algorithm from scratch. We closely followed the algorithm presented in [20]. However, we made specific choices during the implementation process. For the distributions of query points to all the Map nodes, we used the Distributed Cache feature of Hadoop. This query distribution technique is one of weaker links of the method as it assumes relatively small number of query objects, all queries are sent to all Mappers and is not scalable. We would like to address this issue in our future work.

Regarding the dataset, it is not clear in the original paper [20], how general (non-binary) datasets should be pre-processed to work with the algorithm. In our experiments, we found that directly using our test datasets would generate a single bucket. To increase the number of buckets, we pre-processed our dataset applying the standard normal transformation.

In addition, the pre-processing steps were also implemented using MapReduce. As a result, the overall process is composed of three MapReduce jobs. The first one gathers statistics for the transformation, the second one transforms the objects, computes the hash values and filters the objects out and the last one performs the join. While the MapReduce jobs speed up the pre-processing steps, there are some issues as well. In the second job, the bucket of each hash value has to be materialized in memory. Therefore, a larger amount of node memory is needed. In our future work, we would also like to implement a secondary sort comparator in the k-NN join reducer to improve the performance.

When utilizing more hashing tables, some database objects are chosen multiple times. In our measurements, for higher W values the number of processed database objects almost doubled. Authors in the original paper discussed this

topic and their conclusion was that it is not worth it to pre-process and filter unique database data. But, in higher dimensions the similarity computations are more expensive, and this approach should be reconsidered in future work.

3.4 Exact k-NN similarity join approach

In order to be able to evaluate the performance of approximate methods, an exact k-NN similarity join was performed. We used the pivot space approach (Subsection 3.1) with *ReplicationThreshold* parameter set to the number of pivots (thus, all database objects were replicated to all reducers) and the $filter$ parameter explained in the original paper [2] was set to the value 1 (meaning all Voronoi cells C_i are processed on each reducer).

4 Experimental evaluation

In this section, we experimentally evaluate and compare the presented MapReduce k-NN similarity join algorithms. Main emphasis is put on scalability, precision and time complexity of all solutions for high-dimensional data. First, we describe the test datasets and a platform, then we find the best parameters for all the methods and, finally, we compare the performance of all the approaches in multiple testing scenarios.

4.1 Description of datasets and test platform

In the experiments, we perform k-NN similarity joins on three vector datasets with various number of dimensions: 200, 512 and 1000.

The 200 and 1000-dimensional datasets contain histogram vectors which were formed from a few key features located in HTTPS proxy logs collected by the Cisco cloud. Features were transformed into vectors using two techniques. The dataset with 200 dimensions was created by uniform feature mapping into a 4-dimensional hypercube [10]. In the dataset with 1000 dimensions, each HTTPS communication feature was assigned to the closest pre-trained Gaussian utilizing a well known density estimation technique called Gaussian Mixture Model (GMM) [13]. The result vectors are histograms of occurrences of each Gaussian. This feature extraction algorithm is also implemented in the MapReduce framework and is described in detail in the paper [2] and is inspired by works [10] and [14]. The algorithm processes all HTTPS communication features in parallel, groups them by a given key and applies a specific feature transformation strategy to produce final descriptors (vectors).

The last dataset consists of 335944 officially provided key frames from the TRECVid IACC.3 video dataset [1]. The descriptors for each key frame were extracted from the last fully connected layer of the pretrained VGG deep neural network [18] and further reduced to 512 dimensions by the PCA.

All datasets are divided into the database S and query points R. The number of database objects ranges from about $|S|$ =150 000 to 450 000 objects. The size

of the query part ranges from about $|R|$ =180 000 to 320 000 objects in every dataset. Every object contains an ID and a vector of values stored in the space saving format presented in the paper [2]. The size of datasets vary accordingly to dimensions from 0.5GB to 3GB of data in text format. The size of datasets is smaller on purpose because for larger data volumes the LSH method and exact search could not be computed on our cluster due to limitations discussed in Section 3. We employ the Euclidean (L_2) distance metric as the similarity measure.

The experiments ran on a virtualized Hadoop 2.6.0 cluster with 20 worker nodes, each having 6GB RAM and 2 core CPU (Intel(R) Xeon(R) running at 2.20GHz) and were implemented in Java 1.7.

4.2 Fine tuning of experimental methods

In this subsection, we investigate parameters for every tested algorithm. Note that all time values include not only k-NN similarity join job running time but also preprocessing time complexity. The parameter tuning tests ran on the 1000-dimensional dataset and the k value was set to 5.

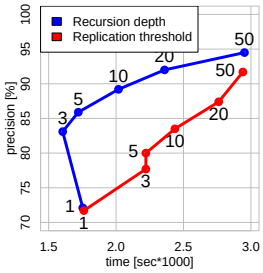

Fig. 2: Pivot-based approach parameters tuning

Fig. 3: Z-curve approach number of shifts tuning

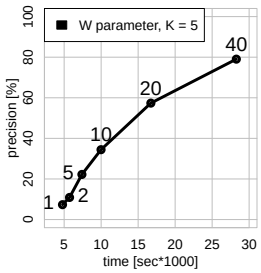

Fig. 4: LSH approach W parameter tuning

In Figure 2, we compare the *ReplicationThreshold* and $MaxRecDepth$ parameters for the pivot-based (Voronoi) approach described in Section 3.1. Although lower parameter values run faster, they don't achieve convincing accuracy. For the rest of the experiments, we fixed $MaxRecDepth$ parameter to the value 10 which promises the best precision and running time trade off. In comparative experiments, we didn't employ the *ReplicationThreshold* at all. In general, the Voronoi space partitioning approach used 2000 randomly preselected pivots, Voronoi cells C_i were grouped into 18 distinct groups G_l and the $filter$ parameter [2] was set to 0.05.

You may notice that total running time for some lower parameter values is longer than for following higher values, e.g. $ReplicationThreshold = 3$ and 5 or $MaxRecDepth = 1$ and 3. Despite more replications, shorter k-NN evaluation

time is caused by the effective candidates processing in the actual algorithm evaluation on each reducer where parent filtering and lower bound filtering techniques in a metric space are utilized [2, 23]. This means the closer k objects to each query are found quicker (the range of a k-NN search is tighter), more candidates are filtered out by the triangle inequality and the total number of actual distance computations is lower.

Figure 3 displays precision and time complexity for the Z-curve approach for growing number of random vector shifts presented in Section 3.2. We may observe that more shifts slightly increases approximation precision, but, on the other hand, running time is prolonged significantly. In other experiments, we fixed the number of shifts to value 5. We used 40 partitions, in order to fit the number of reducers. The Z-curve parameter ϵ was set to 0.008: greater values led to uneven sizes of partitions, whereas smaller values caused reducers in the first phase to die due to the lack of memory (too many objects were sampled). Notice that in the paper [24], different ϵ values did not affect the results.

In Figure 4, we examine the influence of the parameter W to the performance of the LSH method described in Section 3.3. With growing W, both precision and time complexity increase substantially. For other experiments, we fixed W to the value 10 in which the precision is acceptable and running time is comparable to the Z-curve approach. Generally, we used two hash tables (each one containing 20 hashing functions). In the performance tests presented in the paper [20], two to four hash tables were found to produce efficient results. For our data, using more values than two did not significantly alter the performance.

4.3 Comparison of methods

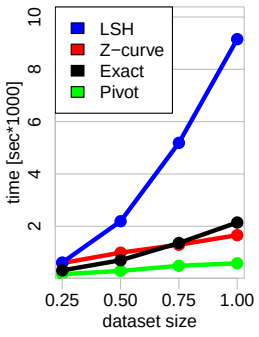

Fig. 5: 200-dimensional dataset: computation time

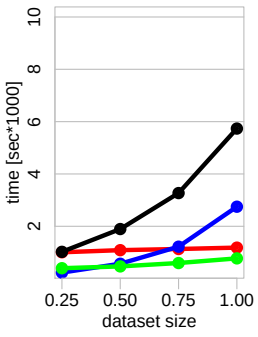

Fig. 6: 512-dimensional dataset: computation time

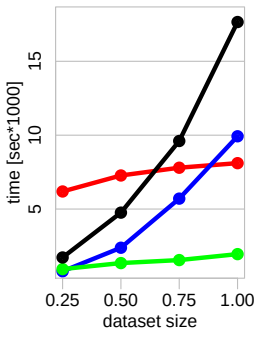

Fig. 7: 1000-dimensional dataset: computation time

We propose multiple testing scenarios designed to test main aspects of each k-NN approximate similarity join algorithm.

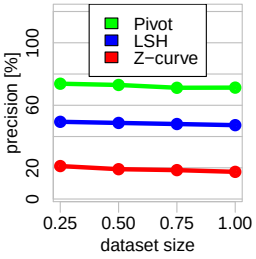

Fig. 8: 200-dimensional dataset: precision

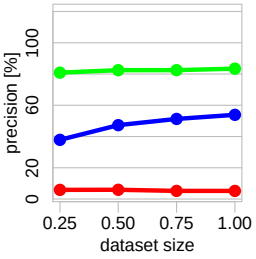

Fig. 9: 512-dimensional dataset: precision

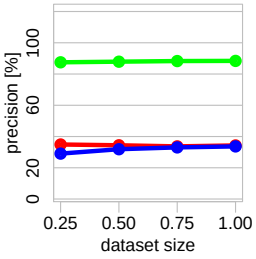

Fig. 10: 1000-dimensional dataset: precision

Size-dependent computation Each of the datasets, both train and test vectors, were sampled in order to create subsets containing $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$ and all of the original data. The methods were tested on each sample. The graphs 5, 6 and 7 show that the running time generally increases with higher dimensionality and dataset size. Observe that for the LSH method, the running time for 200-dimensional dataset is higher than the time for 512-dimensional dataset. The reason for this unexpected behavior is that in the former case few objects were filtered out in the hash-filtering phase. Surprisingly, the Z-curve method is sometimes slower than the exact algorithm. With the exception of the smallest (0.25) samples, the pivot space method shows to be the fastest.

As we can see in the figures 8, 9 and 10, the approximation precision of the methods does not significantly change with the size when each dataset is considered separately. In all cases, the precision of the pivot space method is clearly the highest, ranging from 73% for the 200-dimensional dataset up to 88% for the 1000-dimensional dataset. For 1000 dimensions, the precision of the Z-curve and LSH methods is very similar, fluctuating between 29% and 36%. On the other hand, on the 200 and 512-dimensional datasets the LSH approach outperforms the Z-curve in the precision aspect (about 50% to 20%), but runs substantially longer.

K-dependent computation In the graphs 11 and 12, we investigate the influence of increasing the parameter k (from the k nearest neighbors) to the precision and total similarity join time. All experiments were executed on the 1000-dimensional dataset. In general, the precision stays the same or slowly decreases, whereas time complexity is gradually increasing, but the difference is only marginal. The results of the different approximation methods follow trends identified in the previous graphs. The pivot space approach outperforms other algorithms in both precision and time aspects, the LSH approach presents only a slightly better precision compare to the Z-curve, and the Z-curve approach is the algorithm with the second best execution time. Remember that the LSH approach requires sending all query objects to all mappers. Maintaining the local

k nearest neighbors is quite memory demanding. Due to this limitation we were not able to measure the LSH performance for higher k values on our cluster.

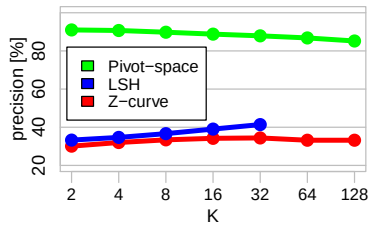

Fig. 11: k-dependent computation: precision

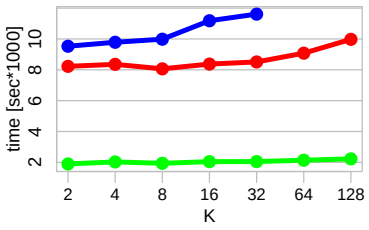

Fig. 12: k-dependent computation: time

4.4 Discussion

In the experiments, three related approximate MapReduce-based k-NN similarity joins on Hadoop were investigated using settings recommended from the original papers. Note that both Z-curve and LSH related papers used mainly just low-dimensional datasets during the design of the approaches (30 dimensions in [24], 32 an 64 dimensions in [20]). In the experiments, the pivot-based approach using the repetitive Voronoi partitioning significantly outperformed the other two methods in both precision and efficiency. Our hypothesis is that for high-dimensional data the Z-curve and LSH methods suffer from the random shifts and hash functions that do not reflect data distributions. We verified this hypothesis on our synthetic 10-dimensional dataset in which all three methods provided expected behavior, as presented in the original papers. Note that specific subsets of the dataset could potentially reside in low-dimensional manifolds. Hence, finetuning specific parameters of the two methods (number of shifts in Figure 3 and W in Figure 4) do not provide a significant performance boost (i.e., for effective retrieval the computational time is extremely long, even longer than naive similarity join). In the future, we plan to investigate both Z-curve and LSH methods more thoroughly and try to design more effective approximate k-NN similarity join strategies.

On the other hand, the Pivot-based approach uses representatives from the data distribution and employs pairwise distances to determine data replication strategies. As demonstrated also by metric access methods for simple k-NN search [23, 17], it seems that the distance-based approach can be also directly used as a robust and intuitive method for approximate k-NN similarity joins in high-dimensional spaces.

5 Conclusions

In the paper, we have focused on approximate k-NN similarity joins in the MapReduce environment on Hadoop. Although comparative studies have been proposed for the considered approaches, the studies focused mainly on low-dimensional data. According to our findings, the dimensionality affects the conclusions about the compared approaches. Two out of three methods previously tested for low-dimensional data did not perform well under their original recommended design and settings.

In the future, we plan to thoroughly analyze and track the bottlenecks of all the methods and try to provide a theoretically sound explanation about the performance limits and approximation errors of all the tested approaches. We also consider implementing algorithms in other MapReduce framework such as Spark and observe performance differences. Findings in the very recent paper [9] promises significant improvements.

Acknowledgments

This project was supported by the GAČR 15-08916S and GAUK 201515 grants.

References

1. Awad, G., Fiscus, J., Michel, M., Joy, D., Kraaij, W., Smeaton, A.F., QuÃľnot, G., Eskevich, M., Aly, R., Jones, G.J.F., Ordelman, R., Huet, B., Larson, M.: Trecvid 2016: Evaluating video search, video event detection, localization, and hyperlinking. In: Proceedings of TRECVID 2016. NIST, USA (2016)
2. Čech, P., Kohout, J., Lokoč, J., Komárek, T., Maroušek, J., Pevný, T.: Feature Extraction and Malware Detection on Large HTTPS Data Using MapReduce, pp. 311–324. Springer International Publishing, Cham (2016)
3. Chavez Gonzalez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Trans. Pattern Anal. Mach. Intell. 30(9), 1647–1658 (Sep 2008)
4. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry. pp. 253–262. SCG '04, ACM, New York, NY, USA (2004)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
6. Ferhatosmanoglu, H., Tuncel, E., Agrawal, D., Abbadi, A.E.: Approximate nearest neighbor searching in multimedia databases. In: Proceedings 17th International Conference on Data Engineering. pp. 503–511 (2001)
7. Giacinto, G.: A nearest-neighbor approach to relevance feedback in content based image retrieval. In: Proceedings of the 6th ACM International Conference on Image and Video Retrieval. pp. 456–463. CIVR '07, ACM, New York, NY, USA (2007)
8. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases. pp. 518–529. VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)

9. Guðmundsson, G.Þ., Amsaleg, L., Jónsson, B.Þ., Franklin, M.J.: Towards engineering a web-scale multimedia service: A case study using spark. In: Proceedings of the 8th ACM on Multimedia Systems Conference, MMSys 2017, Taipei, Taiwan, June 20-23, 2017. pp. 1–12 (2017)
10. Kohout, J., Pevny, T.: Unsupervised detection of malware in persistent web traffic. In: Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on (2015)
11. Lokoc, J., Kohout, J., Cech, P., Skopal, T., Pevný, T.: k-nn classification of malware in HTTPS traffic using the metric space approach. In: Intelligence and Security Informatics - 11th Pacific Asia Workshop, PAISI 2016, Auckland, New Zealand, April 19, 2016, Proceedings. pp. 131–145 (2016)
12. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using mapreduce. Proc. VLDB Endow. 5(10), 1016–1027 (Jun 2012)
13. Marin, J.M., Mengersen, K., Robert, C.P.: Bayesian modelling and inference on mixtures of distributions. In: Dey, D., Rao, C. (eds.) Bayesian ThinkingModeling and Computation, Handbook of Statistics, vol. 25, pp. 459 – 507. Elsevier (2005)
14. Mera, D., Batko, M., Zezula, P.: Towards fast multimedia feature extraction: Hadoop or storm. In: 2014 IEEE International Symposium on Multimedia. pp. 106–109 (Dec 2014)
15. Moise, D., Shestakov, D., Gudmundsson, G.Þ., Amsaleg, L.: Indexing and searching 100m images with map-reduce. In: International Conference on Multimedia Retrieval, ICMR'13, Dallas, TX, USA, April 16-19, 2013. pp. 17–24 (2013)
16. Moise, D., Shestakov, D., Gudmundsson, G.Þ., Amsaleg, L.: Terabyte-scale image similarity search: Experience and best practice. In: Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA. pp. 674–682 (2013)
17. Novak, D., Batko, M.: Metric index: An efficient and scalable solution for similarity search. In: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications. pp. 65–73. IEEE, Washington, DC, USA (2009)
18. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556 (2014)
19. Song, G., Rochas, J., Huet, F., MagoulÃĺs, F.: Solutions for processing k nearest neighbor joins for massive data on mapreduce. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 279–287 (March 2015)
20. Stupar, A., Michel, S., Schenkel, R.: Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In: LSDS-IR (2010)
21. Yao, B., Li, F., Kumar, P.: K nearest neighbor queries and knn-joins in large relational databases (almost) for free. ICDE (2010)
22. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: A unified engine for big data processing. Commun. ACM 59(11), 56–65 (Oct 2016)
23. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, Springer US (2006)
24. Zhang, C., Li, F., Jestes, J.: Efficient parallel knn joins for large data in mapreduce. In: Proceedings of the 15th International Conference on Extending Database Technology. pp. 38–49. EDBT '12, ACM, New York, NY, USA (2012)
25. Zhu, P., Zhan, X., Qiu, W.: Efficient k-nearest neighbors search in high dimensions using mapreduce. In: 2015 IEEE Fifth International Conference on Big Data and Cloud Computing. pp. 23–30 (Aug 2015)