Pivot-based Approximate k-NN Similarity Joins for Big High-dimensional Data[☆]

Přemysl Čech[a], Jakub Lokoč[a], Yasin N. Silva[b]

[a] *Charles University, Faculty of Mathematics and Physics, SIRET Research Group, Malostranské nám, 11800 Prague, Czech Republic*
[b] *Arizona State University, 4701 W. Thunderbird Road, Glendale, AZ 85306, USA*

Abstract

Given an appropriate similarity model, the k-nearest neighbor similarity join represents a useful yet costly operator for data mining, data analysis and data exploration applications. The time to evaluate the operator depends on the size of datasets, data distribution and the dimensionality of data representations. For vast volumes of high-dimensional data, only distributed and approximate approaches make the joins practically feasible. In this paper, we investigate and evaluate the performance of multiple MapReduce-based approximate k-NN similarity join approaches on two leading Big Data systems Apache Hadoop and Spark. Focusing on the metric space approach relying on reference dataset objects (pivots), this paper investigates distributed similarity join techniques with and without approximation guarantees and also proposes high-dimensional extensions to previously proposed algorithms. The paper describes the design guidelines, algorithmic details, and key theoretical underpinnings of the compared approaches and also presents the empirical performance evaluation, approximation precision, and scalability properties of the implemented algorithms. Moreover, the Spark source code of all these algorithms has been made publicly available. Key findings of the experimental analysis are that randomly initialized pivot-based methods perform well with big high-dimensional data and that, in general, the selection of the best algorithm depends on the desired levels of approximation guarantee, precision and execution time.

Keywords: Hadoop, Spark, MapReduce, k-NN, Approximate similarity join, High-dimensional data

1. Introduction

The k-nearest neighbor (k-NN) similarity join is an asymmetric operation that returns the k most similar objects in a dataset S for each query object

[☆] This paper is an extended version of previous papers by Cech et al. [1, 2].

Email addresses: cech@ksi.mff.cuni.cz (Přemysl Čech), lokoc@ksi.mff.cuni.cz (Jakub Lokoč), ysilva@asu.edu (Yasin N. Silva)

in a dataset R. In recent years, the study of k-NN joins attracted considerable amount of attention due to their applicability in various 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 potentially relevant entities for each object in the database. k-NN similarity join applications can be found, for example, in image and video retrieval [3, 4, 5, 6], spatial databases [7], pattern recognition [8], and network communication analysis and malware detection frameworks [2, 9].

Because data volumes are often too large to be processed on a single machine (especially for high-dimensional data), we focus on the distributed MapReduce environment [10] running on Hadoop[1] and Spark[2]. MapReduce is a widely adopted framework and is considered an efficient and scalable solution for distributed big data processing. MapReduce programs are designed to run on large clusters of commodity hardware and employ a programming paradigm similar to the divide and conquer approach. Datasets are loaded, split and pre-processed in the map phase and the main execution and evaluation of an algorithm is performed in parallel on smaller data fractions in the reduce phase.

In this paper, we study approximate k-NN similarity join algorithms that can provide significant speedup compared to the exact similarity join while still preserving high results precision. In many domains, the difference between exact and slightly different k nearest results is acceptable. This is particularly the case in scenarios where computing the exact similarity joins over big high-dimensional data would take significantly large execution times.

Our study focuses on similarity joins for MapReduce environments based on the metric space approach [11]. This approach provides a universal framework for efficient processing of various similarity models. For evaluations on vector data, we also revisited and extended two previously proposed k-NN similarity join approaches designed for vector spaces. In this paper, we focus on algorithms employing data organizations and replication strategies initialized randomly as these techniques can be conveniently applied on Big Data in different domains. Although a study tackling related similarity joins has been previously published for Hadoop [12], the study focused on low dimensional data. The subsequent journal paper [13] tested data up to 386 dimensions and highlighted limitations for most k-NN join methods on such high-dimensional dataset. The need of effective and efficient k-NN similarity joins for high-dimensional data led us to (1) design distributed similarity join techniques with thresholds or approximation guarantees, (2) revise available MapReduce algorithms integrating extensions to more efficiently handle high-dimensional data, (3) consider the implementation of such algorithms on a different platform - Spark (in addition to Hadoop), and (4) experimentally evaluate and compare the performance of the different approaches.

[1] `http://hadoop.apache.org/`
[2] `http://spark.apache.org/`

This paper extends a short conference paper that presented a comparison of our heuristic method with two previously proposed approaches on Hadoop [1] and follows the paper proposing the pivot-based heuristic k-NN join method [2]. This paper significantly extends the previous papers by introducing a new MapReduce based method that supports an ϵ-guaranteed approximation, i.e., an approximate version of the k-NN join where the distance from each query point to its farthest neighbor is constrained in terms of a parameter (ϵ) and the distance to the farthest neighbor in the exact solution. Furthermore, this paper includes the implementation guidelines for Spark and a thorough, and mostly new, set of experimental results.

1.1. Paper contributions

The overall contribution of our work can be summarized into four points:

- Extensions of previously proposed k-NN similarity join algorithms on MapReduce to process big high-dimensional data more efficiently.
- The introduction of pivot-based k-NN similarity join heuristic approaches on MapReduce that support approximation-related thresholds and guarantees. We analyze an approach that provides the ϵ-guarantee (which constrains the distance from each query point to its furthest neighbor returned in the k-NN join). We include a discussion of the theoretical foundations that support the proposed methods.
- The Spark and Hadoop implementation guidelines of the proposed MapReduce join methods. We point out the limitations of different platforms and show why Spark provides faster execution times. We also provide the source code of the Spark implementation of all the evaluated methods, including our new implementations of baseline related approaches based on space filling curves (Z-curve) and locality sensitive hashing.
- Thorough and extensive performance evaluation on large data with different dimensionality (from 10 to 1000 dimensions) running on fully distributed Amazon clusters, with most experiments evaluated on the Spark platform processing up to tens of millions of objects. This analysis provides guidance for selecting an appropriate algorithm for distributed k-NN join based on workload and approximation precision requirements.

The remaining part of the paper is structured in the following way. In Section 2, basic formal definitions and common terms are presented. An overview of similarity join problems, two related methods, and several proposed extensions of these methods are covered in Section 3. Section 4 presents several exact and approximate pivot-based k-NN similarity join algorithms on MapReduce and provides their implementation guidelines. In Section 5, the performance evaluation of all the implemented algorithms is presented and the results are discussed. Section 6 concludes the paper.

2. Preliminaries

The fundamental concepts and basic definitions related to approximate k-NN similarity joins are summarized in the following subsections, considering the standard notations [13, 11].

2.1. Similarity model and k-NN joins

In this work, we address the efficiency of k-NN similarity joins of complex objects obj_i (e.g., images or network traffic snapshots) modeled by high-dimensional vectors $o_i \in \mathbb{R}^n$. Unless otherwise stated, in the following text the term object denotes the vector representation. 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)[3].

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 k-NN 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:

$$kNN(q, S) = \{X \subset S; |X| = k \wedge \forall x \in X, \forall y \in S - X : \delta(q, x) \leq \delta(q, y)\} \quad (1)$$

The k-NN similarity join is defined as:

$$R \ltimes S = \{(q, s) \mid q \in R, s \in kNN(q, S)\}. \quad (2)$$

Because of the high computational complexity of similarity joins, approximations of the joins can be considered to significantly reduce computation costs while maintaining 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 the exact k-NN:

$$\begin{gathered} kNN_a(q, S) = \{X \subset S; |X| = k \wedge \\ \max_{x \in kNN(q,S)} \delta(q, x) \leq \max_{x \in X} \delta(q, x) \leq \epsilon \cdot \max_{x \in kNN(q,S)} \delta(q, x)\} \end{gathered} \quad (3)$$

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

$$R \ltimes_a S = \{(q, s) \mid q \in R, s \in kNN_a(q, S)\}, \quad (4)$$

While the previous definition of the approximate k-NN join has been commonly used in the literature, to the best of our knowledge, no previous paper

[3]The effectiveness of the distance function and feature extraction mapping from *obj* to *o* is the subject of similarity modeling.

has actually implemented a MapReduce k-NN similarity join algorithm that receives ϵ as a parameter and guarantees the ϵ-related property specified in the definition. Instead, previous papers have primarily proposed heuristic methods that aim at (1) having shorter execution times than the exact k-NN join, and (2) having a relatively high result quality quantified by alternative approximation measures like precision and total distance error (these and other measures are presented in Section 2.3). To present a comprehensive study and the trade-offs of different types of approximate k-NN joins, in this paper we present a method that satisfies the ϵ guarantee and compare it with extensions of several heuristic-based methods.

2.2. Metric filtering principles

Since the new methods for distributed k-NN similarity joins are based on the metric space approach [11], we briefly present the fundamental principles of distance based metric indexing for an exact k-NN search. Note that $kNN(q, S)$ corresponds to a metric space ball-region $B = Ball(q, r_q)$, selecting objects from S based on the dynamically estimated radius $r_q \in \mathbb{R}_0^+$. In general, the ball-region defines a set of covered objects $o \in B \subset \mathbb{R}^n$ iff $\delta(q, o) \leq r_q$.

Given a metric space $M = (\mathbb{R}^n, \delta)$, distance based approaches rely on precomputed distances to a set of reference points, so called pivots $P \subset \mathbb{R}^n$. In connection with the triangle inequality property of δ, these distances can be used to efficiently estimate the lower-bound δ_{LB} and upper-bound δ_{UB} distances between a query object $q \in R$ and a database object $o \in S$. Formally, given a pivot $p \in P$ and precomputed distances $\delta(q, p)$ and $\delta(o, p)$:

$$\delta_{LB}(q, o) = |\delta(q, p) - \delta(o, p)| \leq \delta(q, o) \leq \delta(q, p) + \delta(o, p) = \delta_{UB}(q, o). \quad (5)$$

The k-NN query processing is usually designed as an algorithm that maintains and greedily updates the actual set of k closest candidate objects from S, using also the actual query ball radius $r'_q \in \mathbb{R}_0^+, r'_q \geq r_q$. Hence, given the actual query ball $\mathrm{Q} = Ball(q, r'_q)$, an object $o \in S$ can be filtered if

$$r'_q \leq \max_{\forall p \in P} |\delta(q, p) - \delta(o, p)|, \quad (6)$$

without the evaluation of $\delta(q, o)$ which gets costly with high-dimensional data.

The metric space approach provides also two basic data partitioning options for filtering entire groups of objects. The first option is by making use of the already mentioned ball-regions. Given a ball-region based data partition $D = Ball(p, r_p)$ containing selected dataset objects and the actual query ball $\mathrm{Q} = Ball(q, r'_q)$, all objects in the data ball-region D can be filtered if

$$r_p + r'_q < \delta(p, q). \quad (7)$$

The (generalized) hyperplane partitioning represents the second option, which incorporates two pivots. Using pivots $p_1, p_2 \in P$, the metric space is divided

into two sets $S_1 = \{o \mid o \in S, \delta(p_1, o) < \delta(p_2, o)\}$ and $S_2 = \{o \mid o \in S, \delta(p_2, o) \leq \delta(p_1, o)\}$. Given the hyperplane based data partition S_1 and S_2 containing selected dataset objects and the actual query ball $\mathrm{Q} = Ball(q, r'_q)$, all objects in the data partition S_1 can be filtered if

$$\delta(p_1, q) - r'_q \geq \delta(p_2, q) + r'_q. \tag{8}$$

The presented principles are frequently used by various metric access methods for an efficient exact k-NN search [11]. The principles are directly implied from metric axioms. However, in high-dimensional spaces, the distances between dataset objects are often high and similar (the curse of dimensionality effect). Hence, the conditions to safely prune some objects or partitions are mostly not satisfied and the approximate and/or distributed search becomes necessary for more efficient retrieval.

2.3. Approximation measures

Once approximate k-NN joins are considered, the similarity join approximation quality and error have to be evaluated. For this purpose, different approximation measures [14] can be utilized.

The k-NN query approximation precision (or recall) with respect to the exact k-NN search is defined as:

$$precision(k, q, S) = \frac{|kNN(q, S) \cap kNN_a(q, S)|}{k} \tag{9}$$

An object $o_i \in kNN(q, S)$ matches an object $o_j \in kNN_a(q, S)$ iff either $ID(o_i) = ID(o_j)$ or $\delta(q, o_i) = \delta(q, o_j)$ (equal distant objects from a query q may be present in the k-NN result in an arbitrary order). Final approximate precision is computed as a sum of matching k nearest objects divided by the k.

Other approximation measures we use are the total distance ratio,

$$DR(k, q, S) = \frac{\sum_{i=1}^{k} \delta(q, kNN(q, S)[i])}{\sum_{i=1}^{k} \delta(q, kNN_a(q, S)[i])} \tag{10}$$

which represents a ratio between the sums of all neighbor distances in exact and approximate join results and the effective (epsilon) error for the k^{th} neighbor,

$$\epsilon_{eff}(k, q, S) = \frac{\delta(q, kNN_a(q, S)[k])}{\delta(q, kNN(q, S)[k])} \tag{11}$$

comparing the distances to a specific neighbor (usually at the k^{th} position).

In all definitions, the $kNN(q, S)[i]$ expression stands for the i^{th} neighbor in a sorted $kNN(q, S)$ result.

2.4. MapReduce environment

Since data volumes are significantly increasing every day, centralized solutions are often highly inefficient for large data processing. Therefore, the need for effective distributed data processing is emerging. In this paper, we have adopted the MapReduce [10] paradigm that is often used for parallel processing of big datasets. The algorithms described in following sections are implemented in the Hadoop and Spark MapReduce environments which consist 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 their assigned groups 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.

Spark is an in-memory distributed processing engine that uses resilient distributed datasets (RDDs) as its data storage foundation. An RDD is a read-only multiset that is distributed over a computer cluster. Spark has several components that enable different types of data processing, for example Spark Core (supports core data manipulations), Spark SQL (provides support for structured and semi-structured data and a domain-specific query language), and Spark Streaming (supports streaming analytics).

Spark Core is one of Spark's main data processing components. It supports a wide array of distributed data manipulation operations including transformations such as *map*, *filter*, *reduceByKey*, *groupByKey*, *join* and actions such as *reduce*, *collect* and *count*. Most of these operations take RDDs as input and produce RDDs as their output. All Spark transformation functions are evaluated lazily, meaning operations are executed in one optimized data stream after an action function is called. If an RDD is used multiple times, caching techniques can be employed. Spark supports multiple levels of caching, e.g. MEMORY_ONLY persistence (keep all objects or RDDs in main memory) or MEMORY_AND_DISK persistence (prefer main memory but if RDDs don't fit there the rest is saved to hard drives). Spark can run in different setups, e.g. on Hadoop or stand alone. For our testing purposes we used Spark running on Hadoop utilizing Hadoop Yarn (resource manager) and HDFS services.

3. Related work on similarity joins

Many types of different similarity joins have been defined and studied over recent years. Specifically, previous work in this area studied k-Distance joins [15] (returns the smallest k pairs between two datasets), range query joins [16, 17] (returns all the pairs with a distance equal to or smaller than a given threshold) and k-NN similarity joins (for each record of the first dataset, it returns the k closest records in the second dataset) [18, 19]. Some join techniques focus just on specific data types, e.g. set-similarity joins [20, 21] or string joins with edit distance constraints (Ed-Join [22], Trie-Join [23]). Other approaches, like QuickJoin [24], use pivot-based iterative space partitioning or utilize grid structure (Epsilon Grid - EGO [25]). Similarity joins were also studied in the context of database systems and database operators [26, 27]. In this paper, we focus on the last type of similarity joins: the k-NN joins which return the k nearest neighbors for each query object. The k-NN joins are usually specified on either metric spaces [19, 24] or just vector data spaces [28, 29]. Some centralized solutions for k-NN joins employ an index structure providing a significant evaluation speed up, e.g. the R-tree (MuX [18]) or B^+-tree (iJoin [30]) based techniques.

Since sharing a complex index structure is not efficient and perhaps not even viable in a distributed environment those algorithms cannot be optimally parallelized. On the other hand, different algorithms were recently proposed directly for the MapReduce framework which is designed to work in a fully distributed environment composed of up to thousands of computing machines. Specifically, for the MapReduce paradigm, multiple methods were proposed for range query joins [17, 31, 32] and k-NN joins utilizing pivot space partitioning [2, 19, 28, 33], space filling curves (Z-curve [34]), locality sensitive hashing [35, 36] and Hamming distance filtering [29]. The k-NN algorithms (e.g. [34, 35]) usually work in three phases: a data partitioning phase, a partial k-NN join computation, and an intermediate results merge phase. However, not all methods need the third merge phase (e.g. [19, 28]) because the final results are already produced in the second phase.

Related papers [37, 38, 39, 40] have analyzed the advantages, disadvantages and bottlenecks of two of the most well-known distributed MapReduce platforms: Hadoop and Spark. Hadoop is a framework with high focus on disk persisting operations while Spark aims to take advantage of distributed random access memory (RAM) on cluster nodes and stores data on hard drives only when main memory space is insufficient. Since most of the previous work considering similarity k-NN joins has been proposed only for the Hadoop framework, we also integrate the implementation and performance evaluation on the Spark platform. Some of the algorithm implementation details, in fact, significantly differ between these two platforms.

In this paper, we primarily focus on MapReduce-based general metric k-NN similarity joins described in detail later in Section 4. For the experiments and comparisons performed on vector data, we selected and revisited the implementation of two methods for vector spaces: space-filling Z-curve and locality

sensitive hashing. Since the metric joins use randomly selected database objects for indexing, we have selected methods that also use a convenient random initialization of data partitions.

3.1. Space-filling curve based k-NN similarity joins on MapReduce

A space-filling curve is a bijection which maps an object from an $n-$dimensional space to a one-dimensional value, trying to preserve the locality of objects with high probability. For example, the z-order curve creates values (referenced as *z-values*) that can be computed easily by interleaving the binary representation of coordinate values. Different types of space filling curves have been proposed for approximate k-NN search, e.g. Z-curve [41], Z-curve with projections [42] or Hilbert curve [43]. In this paper, we adopt the baseline Z-curve solution for the MapReduce environment [34].

When querying the database, the z-value of a query object is calculated and k database objects with nearest z-values are returned. To reach more precise results, α independent copies of the database and queries are created in the preprocessing phase, each of them shifted by a random vector $v_i \in \mathbb{R}^n$. For each database copy S_i ($\forall o \in S : o_i = o + v_i, S_i = \bigcup_i \{o_i\}$ and for $i = 0, S_0 = S$), z-values of modified objects are computed and sorted. Similarly, all query objects in R are shifted to copies R_i. Each $q_i \in R_i$ is used to query S_i for $2 \cdot k$ objects with the k nearest lower and k nearest higher z-values from S_i to q_i. Thus, up to $2 \cdot \alpha \cdot k$ distinct candidates are collected in total, their distance to the query object is computed and the resulting k nearest candidates are returned.

The centralized solution has been adapted for the MapReduce framework [34]. To distribute the work among the nodes, the objects in each copy S_i and R_i are split in n partitions, depending on their z-value. Every query object belongs to exactly one partition. We must, however, ensure that the partition contains all nearest neighbor candidates. Thus a query object with the maximum z-value needs k database objects with higher z-values copied over (if any exist) in each partition. Analogically, a query object with the minimum z-value needs k database objects with lower z-value. Since the intention is to distribute the objects equally, the best boundary points would be $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$ quantiles of R_i. Nonetheless, precise partitioning may be very expensive due to large data volumes. Instead, objects are sampled and depending on their values and the probability model, approximate quantiles are determined.

Inside each partition, every present query object is used to find $2 \cdot k$ nearest database object candidates. Distances to the candidates are evaluated and intermediate k-NN results are returned. 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. Finally, the nearest objects for each query are detected by merging the candidate k-NN results obtained from all copies S_i.

3.1.1. Implementation revision

The Hadoop application runs in three MapReduce jobs. The Java source code of the MapReduce solution was provided by its authors [34]. We modified it

slightly and adjusted the data structures to fit our object representation and also to be able to load partition objects in memory. For purposes of high-dimensional data computation, we also improved the z-values serialization and implemented the z-value computation of floating point values (values are converted to integers by scaling them by the given constant). The algorithm itself has not been modified.

Our Z-curve implementation in Spark includes few improvements and adjustments. We integrated an option named $OnlyZorder$ which decides how objects with mapped z-value are stored. In case $OnlyZorder = true$ the algorithm works in the same way as Hadoop version while the $OnlyZorder = false$ option means that we store an original object's vector together with the z-value which results in a higher memory usage but the computation is faster because the back transformation from the z-value to original vector coordinates does not have to be performed (and we can also utilize a fast L2 distance for sparse vectors which is used for other methods).

We also integrated a second parameter called $EntirePartitions$ which determines how many database objects in each partition S_i are considered for real distance computations to a query $q \in R_i$. If $EntirePartitions = false$ then the algorithm uses the original $2 \cdot k$ closest neighbors by z-values from an S_i. In case the $EntirePartitions = true$, all database objects in a specific partition S_i are considered and all distances $d(q, s_i), s_i \in S_i$ are evaluated. The second option ($EntirePartitions = true$) leads to higher approximation precision but runs substantially longer. Detailed results for all options are presented in the experimental Section 5.

Other than that, the Spark implementation follows the original Hadoop algorithm and the best Spark programming practices.. All shared data structures are broadcasted to all executors and intermediate results are kept in memory (persistence mode is set to the MEMORY_AND_DISK mode) and, usually, don't have to be written to HDFS. Here we observe the most significant speed up for a comparison between Hadoop and Spark implementations among all studied k-NN join algorithms (concrete numbers are presented in Section 5). This observation is explained by many disk operations performed by the Hadoop Z-curve implementation. Also the Hadoop version is implemented in three separate MapReduce jobs and utilizes the MultipleOutputs class for storing temporary results and statistics (e.g. partitions). These results have to be merged and distributed to all mappers and reducers in following stages which requires a lot of I/O operations. Compared to that, the Spark implementation keeps all needed data in memory and does not require repetitive (expensive) disk accesses (assuming reasonable amount of RAM is available on computing machines).

3.2. Locality sensitive hashing based k-NN similarity joins on MapReduce

Locality Sensitive Hashing (LSH) [44] is another technique that can be used in the context of k-NN heuristic join algorithms. Specifically, Stupar et al. proposed RankReduce [35], a MapReduce-based approximate algorithm to simultaneously process a small number of k-NN search queries in a single MapReduce job using LSH. The key idea behind RankReduce is to use hashing to build an

index that assigns similar records to the same hash table buckets. Zhu et al. [36] proposed an improved version of RankReduce which builds the index in a more efficient way and also compares queries only with database candidates that appear more frequently in the same buckets as queries. However, both related MapReduce techniques are oriented towards long running querying systems. They maintain the database index and the main goal is to provide fast responses to requested k-NN queries. An important property is that they assume that the number of queries is relatively small. On the other hand, many similarity join application scenarios focus on evaluating a lot of queries at once and possibly only once.

Unlike the previously mentioned methods, our LSH approach does not build persisting database indices because we focus mainly on performing independent similarity joins. Nevertheless, our algorithm could be easily adjusted to store hashed database objects for a later use. Also, the previously proposed techniques are not really scalable because they assumed only a small number of queries R in the input which is a big limitation in many application scenarios. Considering this, we implemented a new LSH algorithm. From a high level point of view, our algorithm just compares database and query objects that fall into the same hash bucket after performing hashing operations on both R and S sets. To increase the approximation precision, intermediate k-NN results are produced for several independent sets of hashing functions and the final k-NN join is formed by merging the intermediate results.

The presented method is composed of two main MapReduce jobs in Hadoop: a hashing job including k-NN evaluation and a merging job. During the map phase of the hashing job, both database (S) and query objects (R) are hashed using a set of i hash tables each containing j hash functions of the form $h_{a,B}(v) = \lfloor (a \cdot v + B)/W \rfloor$, where W is a parameter and a and b are constants generated from the p-normal distribution. For every input record $v \in S \cup R$, a set of output keys (buckets) $hash_i$ are evaluated. One $hash_i$ represents a unique string formed from j hash functions corresponding to the hash table i. The map phase emits pairs of the form $(hash_i, v)$. In the reduce phase of the hashing job, local k-NN candidates are computed for a subset of queries and database objects in every bucket identified by the key $hash_i$. In the second MapReduce job, all partial results are loaded, grouped by the query object IDs and global k-NN results for all queries are produced.

3.2.1. Implementation revision

We implemented this algorithm from scratch and initially followed the algorithm presented in [35] until we realized different needs and limitations of the original approach. Regarding the dataset, it is not clear in the original paper [35], how general (non-binary) datasets should be pre-processed to work with the LSH. In our experiments, we found that using our test datasets directly would often generate a single hashing bucket. To increase the number of buckets, we pre-processed our dataset applying the standard normal transformation (a value x_i is transformed to $x'_i = \frac{x_i - \mu}{\sigma}$). In addition, the pre-processing steps were also implemented using MapReduce. As a result, the overall LSH applica-

tion 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 produces the intermediate k-NN results and the last one performs the merge and outcomes global k-NN results.

Our Spark implementation follows the Hadoop one closely. There is no significant difference, data are also transformed using normal standard transformation, all hash tables are broadcasted to all executors, data are grouped by the output of hashing functions and intermediate results are merged to produce final k-NN join results. The only difference is that dimension statistics and intermediate results don't have to be written back to the HDFS and are kept in memory which speeds up the whole application execution.

4. Pivot-based k-NN similarity joins on MapReduce

Pivot-based methods represent a useful generic approach with convenient random initialization, which nevertheless reflect data distribution by dividing a metric space into partitions centered around global objects (pivots) selected from the dataset. The benefits of pivot-based methods have been investigated for k-NN similarity joins on MapReduce in the work of Lu et al. [19]. The authors describe how mappers cluster objects into groups and reducers perform the k-NN join on each group of objects separately. Distance function properties are used to define exact rules for data replication and filtering of non-relevant objects. However, in high-dimensional metric spaces the rules are not sufficiently efficient (the curse of dimensionality effect). In this section, we investigate algorithms for approximate k-NN similarity joins on MapReduce. First, the work of Lu et al. [19] is presented including our comments on revised parts (Section 4.2). Then, we present our additional modifications of the exact search method to obtain a heuristic method in Section 4.3 and we discuss a method with the ϵ-guarantee in Section 4.4. For better clarity, Table 1 summarizes the symbols of frequently used sets in the following subsections.

$S \subset \mathbb{R}^n$	A finite set of database objects
$R \subset \mathbb{R}^n$	A finite set of query objects
$P \subset S$	A finite set of pivots selected from database objects
$C_i \subset \mathbb{R}^n$	Voronoi cell: $\{x \vert x \in \mathbb{R}^n \wedge p_i \in P \wedge \forall_{p_j \in P}(\delta(x, p_i) \leq \delta(x, p_j))\}$
$S_i = S \cap C_i$	Database objects in the Voronoi cell C_i
$R_i = R \cap C_i$	Query objects in the Voronoi cell C_i
$C = \bigcup_{i=1}^{\vert P \vert} \{C_i\}$	Set of all Voronoi cells for the set of pivots P
$G_1, ..., G_m \subset C$	A decomposition of C into m groups of Voronoi cells
$S_i^l \subseteq S_i$	Subset of database objects from S_i replicated to group G_l
$R_i^l = R_i$	All query objects from R_i are replicated to group G_l

Table 1: Symbols of frequently used sets.

4.1. Exact k-NN similarity join approach

The original version of the pivot-based exact k-NN join algorithm [19] (referred to as PGBJ) utilizes a Voronoi partitioning based on the set of preselected global pivots $p_i \in P$ and a metric distance function δ. The algorithm is composed of two main phases: data preprocessing and k-NN join evaluation. The general evaluation workflow of the algorithm is depicted in Figure 1.

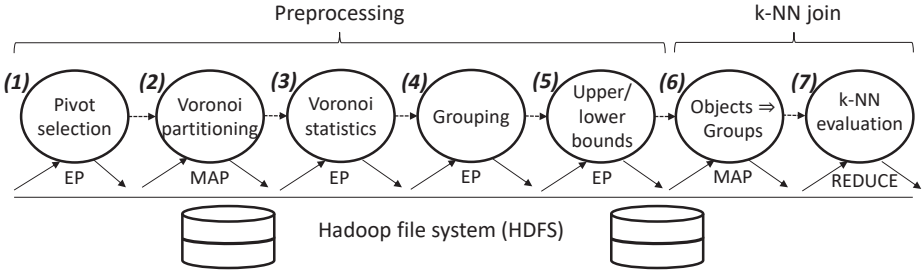

Figure 1: Workflow of the original implementation of the pivot-based approach for MapReduce [19]. Solid arrows represent data flow, dashed arrows represent algorithmic steps.

The preprocessing phase consists of five steps. In step **1.1**, pivots P are selected from the set of database objects S by an external program (EP). In step **1.2**, both sets of database and query objects (S and R) are distributed by a Map job into Voronoi cells C_i according to the preselected pivots $p_i \in P$, forming sets $S_i = S \cap C_i$ and $R_i = R \cap C_i$. Specifically, all distances d_{j_i} from objects $o_j \in S \cup R$ to all pivots p_i ($d_{j_i} = \delta(o_j, p_i)$) are computed and for every object o_j the nearest pivot p_n is identified. For database and query objects, the following records are created and stored (Table 2).

database object record	$rec_o = [o \in S_i, ID_{p_i}, \delta(o, p_i)]$
query object record	$rec_q = [q \in R_i, ID_{p_i}, \delta(q, p_i)]$

Table 2: Records for database and query objects.

In step **1.3**, statistics are computed for every set S_i and R_i including the covering radius, the number of objects o_j and the total size of all objects o_j in the particular cell C_i. Moreover, the distances of the k nearest objects $o \in S_i$ to the pivot p_i are saved for each set S_i for replication rules. In step **1.4**, the Voronoi cells C_i are clustered into bigger disjoint groups G_l to limit the maximum amount of replication (see step **1.6**). The authors proposed an algorithm considering both the geometric and volume properties of cells. Note that every group G_l should contain a similar number of objects to properly balance the next steps of the parallel k-NN join workload. The number of groups G_l should also correspond to the number of reducers (or executors in Spark) used in the next phase. In step **1.5**, the lower and upper bounds for replication that guarantees the correct execution of exact search (see step **1.6**) are computed for

each cell C_i. The following records (Table 3) are created and stored for sets S_i and R_i. The records for whole sets are then distributed throughout the cluster (their size is relative small).

$$\begin{array}{ll} \text{record for } S_i & rec_{S_i} = [|S_i|, size(S_i), lb, ub, \{d_1, ..., d_k\}] \\ \text{record for } R_i & rec_{R_i} = [|R_i|, size(R_i), lb, ub] \end{array}$$

Table 3: Records for sets S_i and R_i including the number of objects, size of objects and the minimal (lb) and maximal (ub) distance from the pivot p_i to objects in the set. rec_{S_i} contains also distances to the k nearest objects to the pivot p_i.

The second phase performs k-NN join of two sets (S and R) in a parallel MapReduce environment (one MapReduce job). In the replication step **1.6**, all query objects $q \in R_i$ are assigned to a group G_l iff $C_i \in G_l$. We will denote the corresponding group in the upper index (R_i^l). Each database object $o \in S$ is assigned to all groups G_l for which a lower bound $LB(o, R_i)$ is less than or equal to an upper bound $UB(R_i)$ for any $R_i \subset C_i \in G_l$. The utilized query radius upper bound $UB(R_i)$ is precomputed in step **1.5** as depicted in Figure 2, considering the distance $R_i.ub$ of the furthest $q \in R_i$ from p_i and the stored distances of the k nearest database objects to each pivot. Observe that considering only the furthest query object leads to a cheaper but less effective replication rule. The lower bound distance to the furthest query object in R_i for one database object o with the closest pivot p_j is determined by the formula:

$$LB(o, R_i) = \max\{0, \delta(p_i, p_j) - R_i.ub - \delta(o, p_j)\}.$$

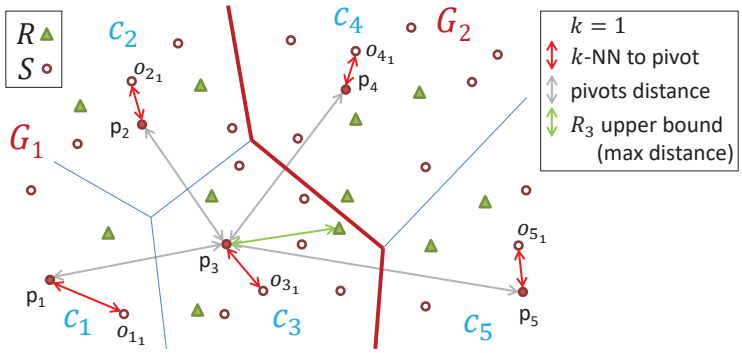

Figure 2: An illustration showing how to compute an upper bound for the Voronoi cell C_3 (query objects in the cell C_3 are denoted as R_3) and given $k = 1$. In this case, the result $UB(R_3) = R_3.ub + \delta(p_3, o_{3_1})$ where $R_3.ub$ is the upper bound (maximal distance to a query object $q \in R_3$ from the pivot p_3) and $o_{3_1} \in S_3$ is the closest database object (1-NN) to the pivot p_3.

Technically, the database object independent value

$$v_{ij} = \max\{0, \delta(p_i, p_j) - R_i.ub - UB(R_i)\}$$

for a whole cell S_j can be precomputed and saved in the preprocessing phase and only the comparison $v_{ij} \leq \delta(o, p_j)$ is performed for each object. A simple lower and upper bounds scheme is visualized in Figure 3. An important note is that when Voronoi cells are aggregated into bigger groups, the values are computed for the whole groups, selecting the minimal value across all R_i in the particular group. Objects $o \in S_i$ replicated to a group G_l form the set S_i^l.

Figure 3: Simple visualization of lower and upper bounds.

Finally, every computing unit cu_l (a reducer or executor corresponding to a group G_l) receives the query objects from all the sets R_i^l and all the sets S_j^l of database objects replicated to group G_l (step **1.7**). Technically, the whole records rec_o, rec_q are sent to cu_l as depicted in Algorithm 1, where the corresponding sets of records are denoted as $RecR_i^l$ and $RecS_j^l$. Then, for every query object $q \in R_i^l$, all sets S_j^l are visited according to distances between the query's nearest pivot and pivots p_j. The k-NN query is evaluated using metric space pruning techniques for each set S_j^l. The authors of the original paper [19] used parent (Equation 6) and cosine law filtering strategies. After all sets S_j^l are processed, the final k-NN result for the query q is produced.

4.2. Revisiting the exact k-NN similarity join

In this paper, we revise some ideas regarding the exact search and the overall MapReduce k-NN join algorithm. We consider the efficient and convenient random pivot selection in step **1.1** given that the benefits of more expensive complex pivot selection techniques have been found with numbers of pivots significantly smaller than the ones used in our work (we usually use thousands of pivots) [45]. In step **1.4**, we implemented a grouping variant which reflects the total data size in addition of the number of objects in each group G_l. In our previous work [2], we showed that such grouping is suitable for space-saving data formats compressing sparse representations.

Regarding the k-NN evaluation step **1.7**, we replaced the sorting of database sets S_i^l with respect to a query q assigned to the group G_l. The new version sorts the sets based on the exact distances between the query q and the pivots p_i unlike the original algorithm which estimated the distances using the closest pivot to q. The new ordering represents a heuristic trying to greedily reduce the actual k-NN range radius and also improve the performance of the employed approximate search heuristic. The authors of the original paper [19] used also

the cosine law for more efficient filtering. However, this technique cannot be applied to all metric similarity functions so we omit this rule. On the other hand, we propose the use of the ball (Equation 7) and hyperplane (Equation 8) filtering strategies successfully employed in metric access methods [11].

Algorithm 1 summarizes the revised PGBJ algorithm, highlighting primarily the k-NN join steps **1.6** and **1.7**. The algorithm closely follows the description presented in the previous subsection. In the preprocessing phase, all support data structures are computed and shared across the cluster (pivots, groups, lower bounds). Specifically, the lower bound values v_{lj} (line 6) store values for whole groups in an ascending order. The function for computing upper bounds is presented in Algorithm 2. This algorithm can compute upper bounds either for one query q or for a whole query cell R_q (the only difference is the part starting on line 4) and is used for determining replications in PGBJ and in the ϵ-guaranteed k-NN similarity join algorithm (Section 4.4). In the map phase of Algorithm 1, the query objects $q \in R$ are sent to the corresponding group and the database objects $o_i \in S$ are distributed to multiple groups G_l by values v_{lj}.

In the reduce phase, the k-NN is computed for all queries using the suggested metric space filtering methods and the exact k-NN results are produced.

4.2.1. Implementation revision

The Hadoop implementation is composed of two MapReduce jobs and was provided by the authors [19]. First, global pivots are chosen, they are stored in HDFS and distributed via the Hadoop distributed cache class. Then, the first MapReduce job computes the distances from all objects in both sets R and S to all pivots, and collects Voronoi cell statistics in the map phase. After that, all statistics are merged together, groups are determined and upper and lower bounds are computed (these operations are implemented separately from any MapReduce job). Statistics, group information and bounds are also distributed throughout the cluster. Then, global k-NN results are computed in the second job. In the map phase, all the objects are assigned to the designated groups (including replication). In the reduce phase, the k-NN computation on a subset of queries and database objects is performed as described previously in Section 4.1.

We also implemented the PGBJ algorithm in the Spark environment. The main difference is that the distributed data structures (e.g. global pivots, records with statistics for all sets, group information) do not have to be stored in the HDFS but are kept in memory and broadcasted to all executors. Also, the entire algorithm does not have to be split into two jobs so the total execution time is faster. Empirical results of the comparison are presented in the evaluation Section 5.

4.3. Heuristic k-NN similarity join approach

Since the PGBJ replication algorithm uses pivot-based upper/lower bounds and the replication strategy is designed for whole Voronoi cells, almost all database objects are replicated to all groups in high-dimensional spaces. With

Algorithm 1 Pivot kNN join(R, S, k)

1: *pivots* P = select pivots from S
2: rec_S, rec_R = partition data to Voronoi cells for R, S according to *pivots*
3: *stats* rec_{S_i}, rec_{R_i} = merge Voronoi statistics
4: *groups* G = group Voronoi cells using *stats* and *pivots*
5: *upperBounds* $UB(R_i)$ = computeUB(R_i, k, P, *stats*, *true*)
6: *values* $v_{lj} = \forall G_l \in G : \min_{R_i \in G_l} \{\delta(p_i, p_j) - R_i.ub - UB(R_i)\}$

7: —*map*—
8: **for** query record rec_q **in** rec_R **do**
9: G_l = get group ID for $rec_q.p_i$
10: output pair $[G_l; rec_q]$
11: **end for**
12: **for** db record rec_o **in** rec_S **do**
//db objects can belong to multiple groups - the replication
13: **for** group ID G_l **in** groups G **do**
14: **if** $v_{li} \leq rec_o.\delta(o, p_i)$ **then** //$\delta(o, p_i)$ is precomputed
15: output pair $[G_l; rec_o]$
16: **end if**
17: **end for**
18: **end for**

19: —*reduce*—
20: **for** groups $G_l \in G$ **do** //all records are grouped by the group id key
21: parse all sets with query records to $RecR^l$ and db object records to $RecS^l$
22: **for** record rec_q including q **in** $RecR_i^l, \forall RecR_i^l \in RecR^l$ **do**
23: compute distances $\delta(q, p_j)$ and sort sets $RecS_j^l$ from the closest
24: $kNN_a(q, S) = \emptyset$
25: r_q = computeUB($rec_q, k, P, stats, false$) //initial query radius
26: **for** $RecS_j^l$ **in** $RecS^l$ **do**
27: **if** $\delta(q, p_j) - r_q \geq rec_q.\delta(q, p_i) + r_q$ **then continue** //Equation 8
28: **if** $\delta(q, p_j) > RecS_j^l.ub + r_q$ **then continue** //Equation 7
29: **for** record rec_o including object o **in** $RecS_j^l$ **do**
30: **if** $|\delta(q, p_j) - rec_o.\delta(o, p_j)| > r_q$ **then continue** //Equation 6
31: update $kNN_a(q, S)$ by o
32: **if** $|kNN_a(q, S)| = k$ **then** $r_q = \max_{x_i \in kNN_a(q,S)} \delta(q, x_i)$
33: **end for**
34: **end for**
35: output $kNN_a(q, S)$
36: **end for**
37: **end for**

the increasing dimensionality, also the filtering rules lose their pruning power. Such a behavior is the consequence of the curse of dimensionality problem [46].

Algorithm 2 ComputeUB(rec_q | R_q, k, P, $stats$ rec_{S_i}, rec_{R_i}, $wholeCell$)

```
1: PQ = ∅   // priority queue in descending order by a distance
2: for pivot p_i in P do
3:     for distance d_j in k distances to p_i in rec_{S_i} do
           //distToPiv is computed either for one query q or whole cell R_q
4:         if wholeCell then
5:             distToPiv = R_q.ub + δ(p_q, p_i)
6:         else
7:             distToPiv = rec_q.δ(q, p_i)
8:         end if
9:         dist = distToPiv + d_j
10:        if PQ.size < k then
11:            add dist to PQ
12:        else if PQ.peek > dist then
13:            remove the first (highest) distance from PQ
14:            add dist to PQ
15:        else
16:            break  //distances d_j are sorted in an ascending distance to p_i
17:        end if
18:    end for
19: end for
20: return PQ.peek
```

In high-dimensional spaces, the distances between pairs of objects are similar and thus pivot-based lower bounds of distances are usually equal to zero, while pivot-based upper bounds are often higher than the highest distance between two objects. This paper extends our previous work [2, 1], where we proposed a heuristic k-NN similarity join method on Hadoop which significantly speeds up the k-NN join time but preserves high approximation precision in the average case. The method is labeled as the pivot approximate k-NN join (PAKJ) and its high level schema is similar to the PGBJ method presented in Section 4.1 (Figure 1). Unlike the PGBJ method, the PAKJ method uses no guarantee thresholds to limit replications and the number of visited sets S_i^l by each query.

The replication step used by PAKJ is inspired by a repetitive (recursive) Voronoi partitioning used by state-of-the-art indexing techniques in metric spaces such as M-Index [47]. In each set S_i, every object o is further identified by the pivot permutation prefix [48] determined by the set of closest pivots to o (instead of a single closest pivot). Given the closest pivots to an object o, the proposed replication heuristic assumes that the object o should be replicated mainly to the groups containing Voronoi cells determined by the pivots (illustrated in Figure 4 for objects $o_1, o_2, o_3 \in S$). A new parameter $MaxRecDepth$ sets a threshold for the number of considered closest pivots for each object $o \in S_i$ (i.e., the limit of recursive splitting of the corresponding cell C_i). In the preprocessing phase (step **1.2**), for every database object $o \in S$ the distances

to all pivots are evaluated and the ordered list of the $MaxRecDepth$ nearest pivots $P^o \subset P$ is stored (in the form of pivot IDs) with object o. The replication heuristic in the beginning of the second phase (step **1.6**) utilizes directly the stored lists of nearest pivots. Specifically, every database object o located in a set S_i is replicated to groups $G_l \subset G$ that contain cells determined by pivots from P^o. Hence, the step **1.5** designed to compute upper/lower bounds can be skipped. Database records are stored in the format displayed in Table 4.

database object record	$rec_o = [o \in S_i, \{ID_p \mid p \in P^o\}, \{\delta(o,p) \mid p \in P^o\}]$

Table 4: Adjusted database records for the PAKJ algorithm.

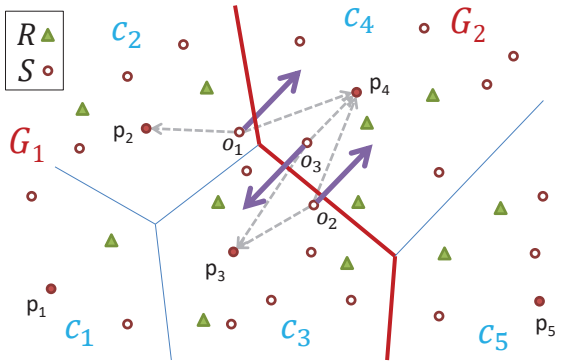

Figure 4: An example of the Voronoi space partitioning and replication of database objects $o_j \in S$. For the $MaxRecDepth = 2$ all three objects o_1, o_2, o_3 near groups boundaries are replicated to the other group because the second closest pivot to the objects o_j lies in the other group.

In the k-NN evaluation step **1.7**, the new parameter called $FilterRatio$ is employed to determine an early stop rule. The $FilterRatio$ parameter represents the percentage of visited cells S_i^l after which the k-NN search is stopped (e.g. $FilterRatio = 0.01$ means that after visiting $0.01 \cdot |P|$ cells the k-NN search is terminated if at least k objects were found).

Our implementation of the PAKJ algorithm in Hadoop and Spark is similar to the PGBJ join. New parameters during replication step and k-NN query processing (early termination) are employed and also upper/lower bounds do not have to be computed because they are not used for the replication strategy.

4.4. k-NN similarity join approach with the $\epsilon-$guarantee

The PAKJ heuristic does not provide the worst case guarantees. This means that despite a good performance in the average case, some k-NN queries forming the join could result in a high effective (epsilon) error (Definition 11).

In this section, we investigate a MapReduce based k-NN similarity join method for metric spaces that tackles the $\epsilon-$guaranteed approximation. Enforcing the $\epsilon-$guarantee to support the approximately correct nearest neighbor

(AC-NN) queries has been presented in the paper by Ciacccia et al. [49, 50]. More specifically, given a query object $q \in R$ and the distance to its nearest neighbor $r_q \in \mathbb{R}_0^+$ the AC-NN query can return any object $o \in S$ such that $\delta(q, o) \leq \epsilon \cdot r_q$, $\epsilon \geq 1$. The authors present an exact nearest neighbor search algorithm that can be adapted for AC-NN queries by substituting the distance to the actual nearest neighbor candidate r_x by $\frac{r_x}{\epsilon}$. The same idea can be applied for approximate k-NN similarity join methods on MapReduce with the $\epsilon-$guarantee. In the following subsection, we summarize a sound formal background [50] clarifying the correctness of the utilized approach for pivot-based k-NN similarity joins on MapReduce.

4.4.1. Formal background

The goal of the following theorems is to show that utilizing the $\frac{r_x}{\epsilon}$ radius for k-NN search in metric spaces preserves the $\epsilon-$guarantee (Definition 3). Moreover, correct metric space filtering methods (object, ball and hyperplane filtering) can be used to speed up approximately correct k-NN queries.

For all the following Theorems, we work with the next premises.

Premises. *Let us assume a given metric space $M = (\mathbb{R}^n, \delta)$, a query object $q \in \mathbb{R}^n$, a finite data set $S \subset \mathbb{R}^n$, an arbitrary candidate result set $X \subset S$, $|X| = k$, an actual query radius $r_x = \max_{x_i \in X} \delta(q, x_i)$, an approximation parameter $\epsilon \in \mathbb{R}^+, \epsilon \geq 1$ and a set of pivots $P \subset S$.*

Theorem 1 clarifies the idea that database objects $o_i \in S - X$ in the annulus $< \frac{r_x}{\epsilon}, r_x >$ centered in q can be skipped and the ϵ-guarantee will not be violated.

Theorem 1. *Let $Y = \{o_i \in S \mid \delta(q, o_i) \geq \frac{r_x}{\epsilon}\}$ and $r_{xy} = \max_{x_i \in kNN(q, X \cup Y)} \delta(q, x_i)$, then it holds that $\frac{r_x}{r_{xy}} \leq \epsilon$.*

Proof 1. *Trivial. $r_{xy} \geq \frac{r_x}{\epsilon}$, which implies that $\epsilon \geq \frac{r_x}{r_{xy}}$.*

Note 1. *As a consequence of Theorem 1, skipping any database object $o_i \in S - X$, $\delta(q, o_i) \geq \frac{r_x}{\epsilon}$ during query processing does not violate the $\epsilon-$guarantee condition for $\epsilon-$approximate k-NN search. Nevertheless, objects o_i satisfying $\delta(q, o_i) < r_x$ can be used to update the actual query radius r_x and improve filtering efficiency.*

Furthermore, Theorems 2, 3, and 4 formalize the idea that filtering techniques defined in Subsection 2.2 (Definitions 6, 7, 8) do not violate the $\epsilon-$guarantee.

Theorem 2. *Let $Y = \{o_i \in S \mid \frac{r_x}{\epsilon} \leq \max_{\forall p \in P} |\delta(q, p) - \delta(o_i, p)|\}$ and $r_{xy} = \max_{x_i \in kNN(q, X \cup Y)} \delta(q, x_i)$. Then it holds that $\frac{r_x}{r_{xy}} \leq \epsilon$.*

Proof 2. *From the triangle inequality and definition of Y it holds that $\delta(q, o_i) \geq \max_{\forall p \in P} |\delta(q, p) - \delta(o_i, p)| \geq \frac{r_x}{\epsilon}$, $\forall o_i \in Y$. In connection with Theorem 1 it holds that $\epsilon \geq \frac{r_x}{r_{xy}}$.*

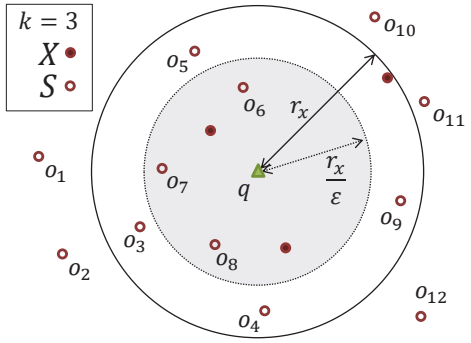

Figure 5: An illustration of Theorem 1. Objects outside of the gray area can be safely pruned.

Theorem 3. *Let* $p \in P$, $r_p \in \mathbb{R}^+$, $Y = \{o_i \in S \mid o_i \in Ball(p, r_p) \wedge r_p + \frac{r_x}{\epsilon} < \delta(q,p)\}$ *and* $r_{xy} = \max_{x_i \in kNN(q, X \cup Y)} \delta(q, x_i)$. *Then it holds that* $\frac{r_x}{r_{xy}} \leq \epsilon$.

Proof 3. *From the triangle inequality, ball region and* Y *definition it holds that* $\delta(q, o_i) \geq \delta(q,p) - r_p > \frac{r_x}{\epsilon}$, $\forall o_i \in Y$. *In connection with Theorem 1 it holds that* $\epsilon \geq \frac{r_x}{r_{xy}}$.

Theorem 4. *Let* $p_1, p_2 \in P$, $S_1 = \{o_i \in S \mid \delta(p_1, o_i) < \delta(p_2, o_i)\}$, $Y = \{o_i \in S_1 \mid \delta(q, p_1) - \frac{r_x}{\epsilon} \geq \delta(q, p_2) + \frac{r_x}{\epsilon}\}$ *and* $r_{xy} = \max_{x_i \in kNN(q, X \cup Y)} \delta(q, x_i)$. *Then it holds that* $\frac{r_x}{r_{xy}} \leq \epsilon$.

Proof 4. *From the definition of partitions* S_1 *and* S_2 *and the set* Y *it holds that* $\delta(q, o_i) \geq \delta(q, p_2) + \frac{r_x}{\epsilon} \geq \frac{r_x}{\epsilon}$, $\forall o_i \in Y$. *Again, in connection with Theorem 1 it holds that* $\epsilon \geq \frac{r_x}{r_{xy}}$.

4.4.2. The $\epsilon-$guaranteed pivot method

Following the theorems 1-4, we present an algorithm called the pivot epsilon guaranteed k-NN join on MapReduce (PEGKJ). It follows the revisited exact k-NN join algorithm (Section 4.2) and basically uses less strict rules (employing the parameter ϵ) during replication and k-NN query evaluation phases.

We follow the schema in Figure 1 with several changes involving mainly the ϵ parameter. The first change is that the upper bound $UB(R_i)$ (estimate of a query radius) for replication of objects to the group containing queries in R_i is divided by the ϵ parameter. As the original $UB(R_i)$ radius estimate guarantees at least k found objects possibly from more than one group, the PEGKJ method computes $UB(R_i)$ using only the sets from the group containing R_i. It is important to realize that computing $UB(R_i)$ from all cells (i.e., from all groups) and lowering the $UB(R_i)$ radius estimate by ϵ could result into a situation where objects used to compute the original $UB(R_i)$ are not replicated to the group containing R_i. Hence, k candidates for a query $q \in R_i$ and initial radius $UB(R_i)$ does not have to be present in the group containing R_i. Starting with a higher initial radius does not help as it could lead to the violation of the

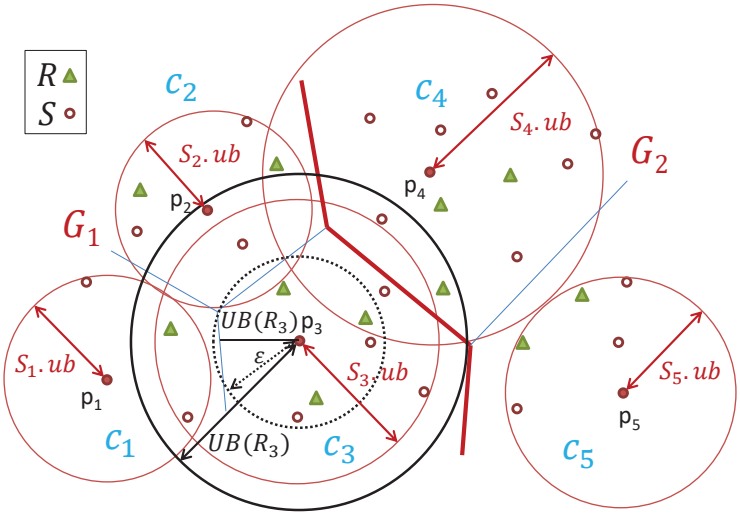

Figure 6: An example demonstrating the ϵ upper bound influence. Considering the queries upper bound for the set R_3 equals to $UB(R_3)$ (the black ball), for the exact k-NN join algorithm some database objects from the overlapping set S_4 are replicated to the group G_1 (depending on the distance of an object o_j to it's corresponding pivot). After the $UB(R_3)$ bound is lowered by the ϵ (the dashed circle) parameter, objects from the set S_4 are no longer replicated.

k-NN $\epsilon-$guarantee for q because the approximate k^{th} nearest neighbor could be located further than the ϵ times the distance to the exact k^{th} nearest neighbor. Specifically, Algorithm 2 skips pivots determining cells from different groups and in the last line of Algorithm 2 the $PQ.peek/\epsilon$ value is returned. An example and the effect of lowering $UB(R_i)$ is depicted in Figure 6.

During the replication phase (potentially) less database objects $o \in S$ are distributed and replicated among the groups because the pre-computed values v_{ij} are greater ($UB(R_i)/\epsilon$ is subtracted). Finally, in the k-NN evaluation phase the ϵ parameter is utilized once first k nearest neighbor candidates are obtained for initial radius estimate $UB(R_i)$. The initial query radius is computed in the same way as in the original paper (using Algorithm 2 where the ϵ is not employed to guarantee a candidate set of size k). However, when k candidates are obtained, the actual radius $\frac{r_q}{\epsilon}$ is correctly used for further query processing (Algorithm 1 lines 27, 28, 30) based on previously presented Theorems 2, 3, 4. Overall, the PEGKJ algorithm is ϵ-correct because all replication and filtering techniques utilize rules preserving the ϵ-guarantee from Definition 3.

In the experiments, we observed that precision drops significantly with the growing ϵ parameter for the PEGKJ method (results are presented in Section 5). This behavior is the consequence of too reduced actual query radius, independently on the filtering rule. However, intuitively replication rules or metric ball-ball overlap tests are less sensitive to approximation than for example parent filtering. To increase precision of PEGKJ, we propose a parameter called

ExactParentFiltering (or shortly *ExactPF*) which determines whether the parent filtering rule (Algorithm 1, line 30) considers $\epsilon-$approximation radius $\frac{r_q}{\epsilon}$ or uses the actual radius r_q.

We also observed that the PGBJ and PEGKJ algorithms tend to replicate almost all database objects to all groups in high-dimensional spaces. This trend is noticeable in our experiments (more in Section 5) but it could be also justified theoretically in the following Theorem 5.

Theorem 5. *Given a metric space $M = (\mathbb{R}^n, \delta)$, a database $S \subset \mathbb{R}^n$ and queries $R \subset \mathbb{R}^n$ partitioned by pivots P to Voronoi cells. Assuming that $UB(R_i)$ is always estimated just from the cell C_i (using $R_i.ub + \delta(p_i, o_{i_j}), o_{i_j} \in S_i$), then $LB(o, R_i) \leq UB(R_i)$ for all objects $o \in S$ and sets $R_i \in R$ iff $\delta_{max} \leq 4 \cdot \delta_{min}$, where $\delta_{max} = \max_{\forall o_i, o_j \in S \cup R} \{\delta(o_i, o_j)\}$ and $\delta_{min} = \min_{\forall o_i, o_j \in S \cup R} \{\delta(o_i, o_j)\}$.*

Proof 5. *$LB(o, R_i) = \delta(p_i, p_j) - R_i.ub - \delta(o, p_j)$, $UB(R_i) = R_i.ub + \delta(p_i, o_{i_j})$. The best case for limiting replications would be $LB(o, R_i) = \delta_{max} - \delta_{min} - \delta_{min}$ and $UB(R_i) = \delta_{min} + \delta_{min}$. Hence, if $\delta_{max} \leq 4 \cdot \delta_{min}$ then $LB(o, R_i) \leq UB(R_i)$ for all database objects and sets R_i.*

Note 2. *Considering $\delta_{max}, \delta_{min}$ from $S \cup R$ is even more optimistic assumption for limiting replications than incorporating $\delta^S_{max}, \delta^S_{min}$ detected just from S as $\delta^S_{max} \leq \delta_{max}, \delta^S_{min} \geq \delta_{min}$ and δ^{RS}_{min} detected just for pairs from $R \times S$ as $\delta^{RS}_{min} \geq \delta_{min}$. If $\delta_{max} \leq 4 \cdot \delta_{min}$ then also $\delta^S_{max} \leq 2 \cdot \delta^S_{min} + 2 \cdot \delta^{RS}_{min}$. We used $S \cup R$ to provide a clue to assess replication for general distance distributions.*

Note 3. *In high-dimensional spaces, all distances between database objects are relatively similar and the probability that $\delta(p_i, p_j)$ is four times greater than the other involved distances in the replication rule is very low. Actually, $LB(o, R_i) = 0$ and values $v_{ij} = 0$ were observed almost always for all datasets in our experiments (including the ϵ-approximation). In such cases, most database objects are replicated to all groups.*

5. Experimental evaluation

In this section, we experimentally evaluate and compare the presented MapReduce k-NN similarity join algorithms. The main emphasis is put on scalability, precision and the overall execution time of all solutions for high-dimensional data. First, we describe the test datasets and the evaluation platform, then we compare selected methods on two MapReduce frameworks, where we present benefits of Spark. For Spark, we investigate parameters for all the presented methods and, finally, we compare the performance of selected approaches in multiple testing scenarios. We have also published all the Spark source-codes in a publicly accessible repository on Github[4].

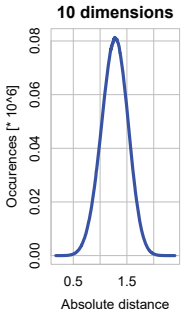

Figure 7: Distance distribution - 10D

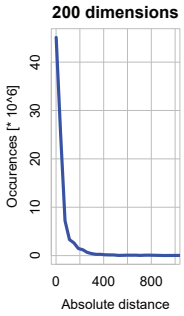

Figure 8: Distance distribution - 200D

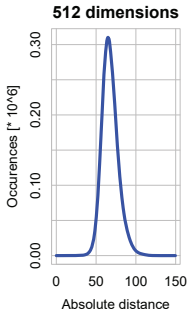

Figure 9: Distance distribution - 512D

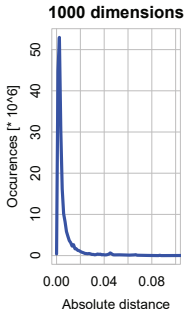

Figure 10: Distance distribution - 1000D

5.1. Description of datasets and test platform

In the experiments, we perform k-NN similarity joins on four vector datasets with various number of dimensions: 10, 200, 512 and 1000. In the experiments, we often refer to the datasets by their unique dimensionality and the concatenated letter "D" (see Figures 7, 8, 9, and 10).

The 10-dimensional dataset is a synthetic dataset made by generating 200,000 objects into a uniform $[0, 1]^{10}$ cube. This artificial dataset was created to investigate and present the performance of compared methods on data with a lower dimensionality.

The 200 and 1000-dimensional datasets contain histogram vectors modeling HTTPS communication (e.g., from web proxy logs). Only high-level communication features of HTTPS requests [51] were aggregated into vectors using two techniques. The dataset with 200 dimensions was created by uniform feature mapping into a 4-dimensional hypercube [51]. In the dataset with 1000 dimensions, HTTPS communication was modeled using the Gaussian Mixture Models approach (GMM) [52]. For more details, see the journal [53], where also the feature extraction algorithm is presented and implemented using the MapReduce framework. 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 512-dimensional dataset consists of 335,944 selected keyframes from the TRECVid IACC.3 video dataset [54]. The descriptors for each key frame were extracted from the last fully connected layer of the pretrained VGG deep neural network [55] and further reduced to 512 dimensions using PCA.

All datasets are divided into the database S and query points R. The distance distributions on a smaller sample for all datasets are presented in Figures 7, 8, 9, and 10. The number of database and query objects ranges in hundreds of thousands objects for most of the experiments. Only the growing data size experiment run on tens of millions of 200D objects. Every object contains a unique

[4]https://github.com/PremyslCech/kNN-joins-spark

object ID and a vector of values stored in the space saving format presented in [2]. The size of the datasets vary according to the number of dimensions from 0.5GB to 5GB of data in the space saving format. We employ the Euclidean (L_2) distance as the similarity measure.

The experiments run on a fully distributed Amazon clusters under the Elastic MapReduce and EC2 services. We used clusters of 5 to 20 computing nodes each containing the Intel Xeon processor having 4 Cores (8 threads) running at 2.5 Ghz, 15 GB RAM and 2x40 GB SSD disk (the Amazon m3.xlarge instance). Data were stored in the S3 storage system.

5.2. Hadoop vs Spark

In the first set of experiments, we analyze the two most popular MapReduce platforms Hadoop and Spark. We compare the platforms on three different similarity k-NN join algorithms – PAKJ and the algorithms based on Z-curves (Section 3.1) and locality sensitive hashing (Section 3.2). In Figures 11, 12 and 13, we compare the join evaluation time on both platforms for all datasets given the same method settings (i.e., the join result was exactly the same on both platforms). Graphs reflect the join evaluation time for the given parameter setup for each method, but similar outcomes were observed for different settings of parameters. All the tested algorithms run faster on Spark because intermediate results and shared support data structures do not have to be serialized and stored in HDFS but are kept in memory for the whole algorithm execution. Also all k-NN join methods are implemented on Hadoop in multiple MapReduce jobs (two or three) and results from previous jobs have to be written back to HDFS and sometimes even merged or otherwise manipulated. The biggest difference is noticeable for the Z-curve algorithm which uses multiple I/O operations on Hadoop to provide proper partitioning and data pre-processing. On Spark, all operations run in memory and temporary data structures do not need to be saved on disk. Because of the convincing results on Spark for our datasets, all other experimental analysis, evaluations and tests are presented just for the Spark platform.

5.3. Fine tuning of k-NN join methods

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

In Figure 14, we study the influence of the number of groups G and randomly selected pivots P used for the Voronoi partitioning on the PAKJ algorithm performance. According to our internal test cases, other pivot selection techniques in the original source codes provided by [19] run substantially longer and did not present any significant improvements for further evaluation. Based on the results, we have fixed the number of pivots to 2,000 and the number of groups to 20 in the remaining experiments. It is suggested to set the number of groups

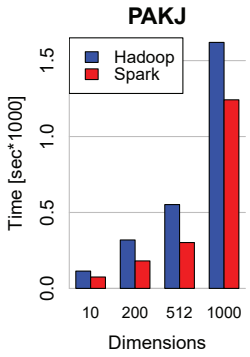

Figure 11: PAKJ approach - join evaluation time Hadoop vs Spark

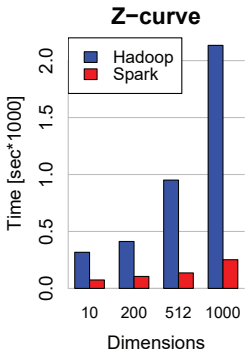

Figure 12: Z-curve approach - join evaluation time Hadoop vs Spark

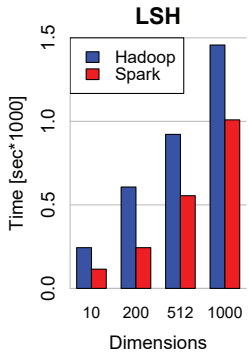

Figure 13: LSH approach - join evaluation time Hadoop vs Spark

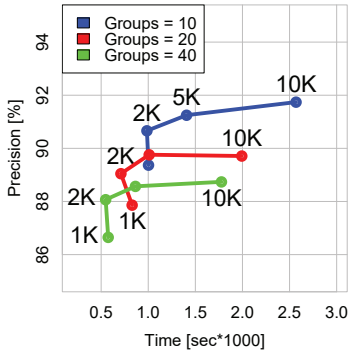

Figure 14: PAKJ - growing number of pivots and groups, $MaxRecDepth$ = 10, $Filter$ = 0.01

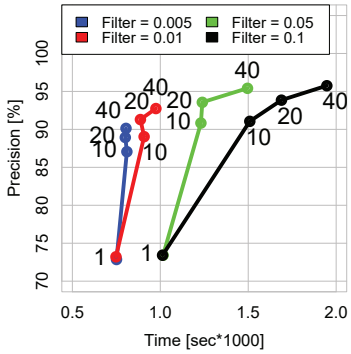

Figure 15: PAKJ - $MaxRecDepth$ and $FilterRatio$ parameters tuning, Pivots = 2000, Groups = 20

to match the number of reducers or executors to achieve the best parallel computation balance.

In Figure 15, we compare the $MaxRecDepth$ and $FilterRatio$ parameters for the PAKJ method (Section 4.3). Although lower parameter values run faster, they achieve also limited accuracy. For the rest of the experiments, we fixed $MaxRecDepth$ parameter to the value 10 (if not specified otherwise) which promises a competitive precision and running time trade off for comparisons with methods based on Z-curves and LSH. The $FilterRatio$ parameter was fixed to the values 0.01 or 0.05.

Observe that the total k-NN join evaluation time for some lower parameter values is longer than for a bit higher values, e.g. $MaxRecDepth$ = 10 and 20 for the $FilterRation$ = 0.01. Despite more replications, shorter k-NN evaluation time is caused by the efficient candidate processing on each executor where

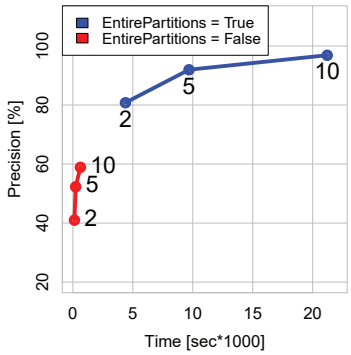

Figure 16: Z-curve - number of shifts parameters tuning, All in memory

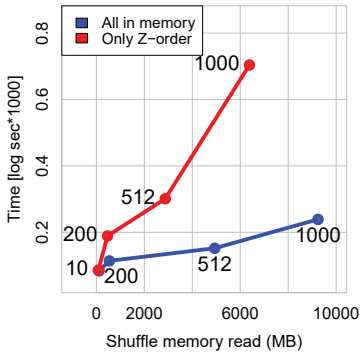

Figure 17: Z-curve - effect of storage options, $Shifts = 5$, $EntirePart = false$

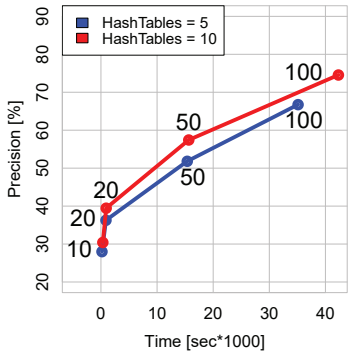

Figure 18: LSH approach - W parameter tuning, $HF = 20$

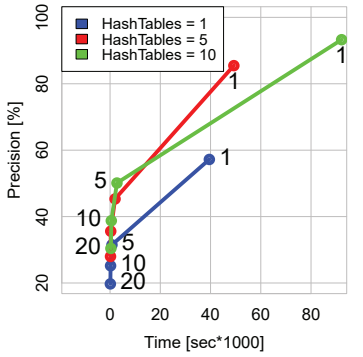

Figure 19: LSH approach - the effect of the number of hash functions. $W = 10$

filtering techniques in a metric space are used (Section 2.2). Note that closer k objects to many query points appear in their group and so the ranges of k-NN queries get tighter. Hence, more candidates can be filtered out, which was revealed also by a lower number of evaluated distance computations.

Figure 16 displays the precision and the overall k-NN join evaluation time for the Z-curve approach for growing number of random vector shifts presented in Section 3.1. We can observe that more shifts slightly increases approximation precision, but the running time is extended significantly. Also, the difference between the $EntirePartitions = true$ and $false$ options is substantial offering big variety of the precision/running time trade-off. In other experiments, we usually fixed the number of shifts to 5. We used 20 partitions, in order to fit the number of executors. The Z-curve sampling rate parameter was set to 0.005 which influences mainly the partitions balance determining a proper level of parallelism.

Figure 17 displays the k-NN join evaluation time and shuffle memory usage for different levels of object storage settings described in subsection 3.1.1. While the "All in memory" option is clearly the faster option, for larger dataset it might not be viable (stored objects might not fit in memory). The "Only Z-order" approach translates Z-order back to the original coordinates and, thus, is considerably slower.

In Figure 18, we examine the influence of the parameter W on the performance of the LSH method described in Section 3.2. With growing W, both precision and time increase substantially. Longer running time for higher W values is mainly caused by hashing objects into bigger buckets (more objects have to be processed by the k-NN join in a large bucket). However, this parameter heavily depends on the specific dataset. For other experiments, we used $W = 10$ to 20 for the 10-dimensional dataset, $W = 1$ to 5 for the 200-dimensional dataset, $W = 100$ or 200 for the 512-dimensional dataset and $W = 20$ for the 1000-dimensional dataset. Generally, we used 10 hash tables each containing 20 hash functions. These parameters are analyzed in Figure 19. With growing number of hash functions both running time and precision drops because more hash bins are generated and queries do not meet all nearest objects in a bucket.

5.4. Approximate pivot methods comparison

In this subsection, we study different parameters and performance of both pivot-based approximate algorithms: PAKJ (the pivot approximate (heuristic) algorithm) and PEGKJ (the pivot epsilon guaranteed approach). All presented graphs were measured on 512D and 1000D datasets and for $K = 10$.

In the first four Figures 20, 21, 22, and 23, we analyze the influence of the PEGKJ parameter $ExactParentFiltering$ ($ExactPF$) on precision, k-NN join evaluation time and effective error for the growing ϵ parameter. Observe that the $ExactPF = true$ option is significantly superior in terms of precision but its running time is higher for increasing ϵ (less database objects are pruned and more distance computations are needed). Nevertheless, for $\epsilon = 5$ the precision of the variant $ExactPF = true$ is higher than the precision of the variant $ExactPF = false$ for $\epsilon = 2$, while their time is similar. Both the maximal and average effective error is also smaller for the $ExactPF = true$ parameter. Based on these observations, we conclude that once high ϵ is acceptable, the variant $ExactPF = true$ represents a better approximation option.

In Figures 24, 25, 26, 27, 28 and 29, we compare different aspects of all approximate pivot-based methods. The $ExactPF$ parameter was set to $true$ for the PEGKJ method and experiments run on 512 and 1000-dimensional datasets. In Figures 24 and 25 you may observe precision/running time trade-off for all methods for different setup of parameters. To highlight one method, PAKJ with $Filter = 0.1$ is closing to 100% precision and is significantly faster than other variants/methods reaching high precision. In general, the PEGKJ algorithm is the slowest method and replicates objects (Figure 26) to almost all groups G (the reasons are explained in Section 4.4) but provides approximation guarantees (Figure 27). The replications directly affect transfered volumes of data (less is better), thus minimizing them enables a method to process larger datasets.

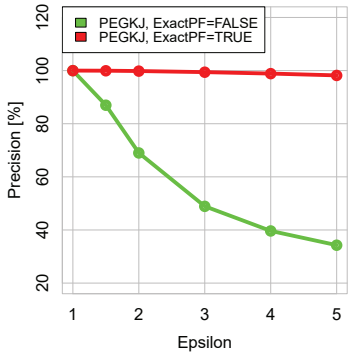

Figure 20: PEGKJ Exact parent filtering - Precision

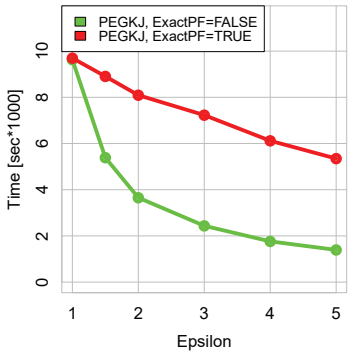

Figure 21: PEGKJ Exact parent filtering - Running time

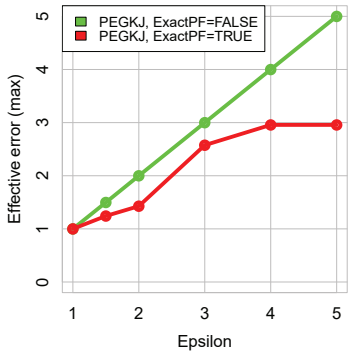

Figure 22: PEGKJ Exact parent filtering - Maximal Effective error

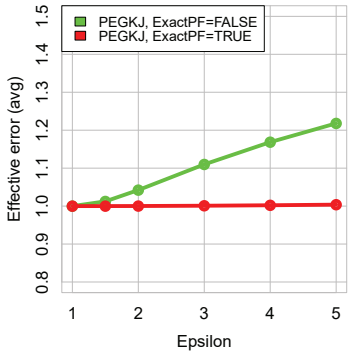

Figure 23: PEGKJ Exact parent filtering - Average Effective error

In the last two graphs 28 and 29, we present cumulatively the number of query objects reaching an effective (real) epsilon error presented on the X-axis. Here the PEGKJ algorithm has the most objects with small effective error for lower $\epsilon = 4$, which means that more approximate k^{th} nearest neighbors from different queries are closer to exact results. Nevertheless, PAKJ methods are following PEGKJ results closely and PEGKJ for higher $\epsilon = 10$ is even worse for most queries than all presented PAKJ approaches on the 1000-dimensional dataset. Average and maximal effective errors for selected methods are presented in Table 5.

We conclude that all pivot-based algorithms perform well under different conditions and a specific algorithm utilization must be decided based on the desired approximation guarantee or an average approximation precision and running time preference.

Method/variant	512D AVG	512D MAX	1000D AVG	1000D MAX
PAKJ, Filter=0.01, Depth=20	1.01329	1.95188	1.01095	6.23379
PAKJ, Filter=0.1, Depth=20	1.00400	1.81889	1.00681	6.21953
PAKJ, Filter=0.3, Depth=20	1.00371	1.81889	1.00615	6.21953
PEGKJ, EPF=true, Epsilon=4	1.00039	1.27834	1.00223	2.95698
PEGKJ, EPF=true, Epsilon=10	1.00735	1.85962	1.01325	4.48893

Table 5: Average and maximal effective (real) errors for pivot-based methods.

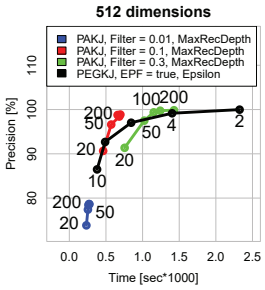

Figure 24: Pivot-based methods - Precision vs Time, 512D

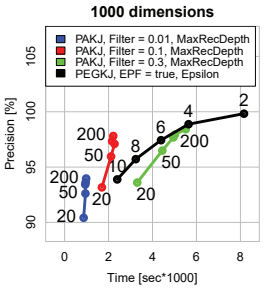

Figure 25: Pivot-based methods - Precis. vs Time, 1000D

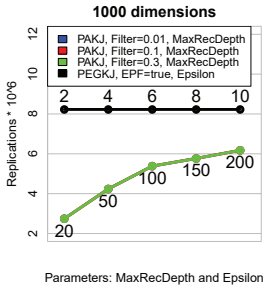

Figure 26: Pivot-based methods - Replications, 1000D

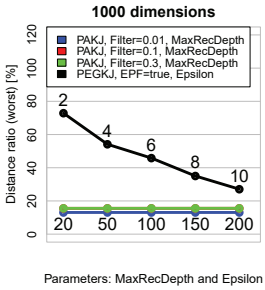

Figure 27: Pivot-based methods - Distance ratio, 1000D

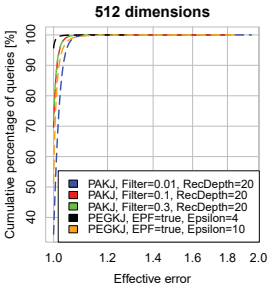

Figure 28: Pivot-based methods - cumul. Eff. error, 512D

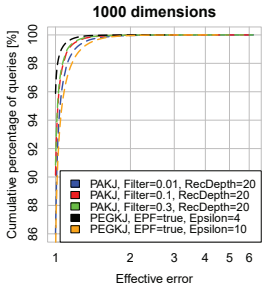

Figure 29: Pivot-based methods - cum. Eff. error, 1000D

5.5. Comparison of pivot-based approach with related approaches

We propose multiple testing scenarios designed to compare the main aspects of pivot-based, Z-curve and LSH k-NN approximate similarity join algorithms. Please note that all the methods use a convenient random initialization of data partitioning.

5.5.1. Precision and k-NN join evaluation time

First, we analyze the performance of all the compared methods on all datasets in Figures 30 and 31. Parameters were set as follows. PAKJ: $MaxRecDepth =$

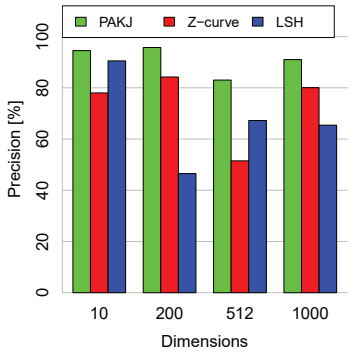

Figure 30: Precision for heuristic methods for all datasets

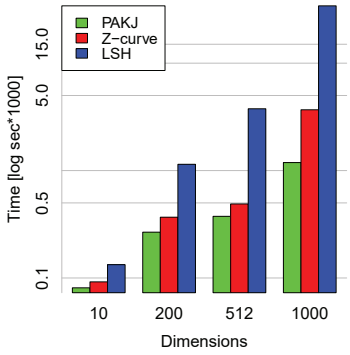

Figure 31: Running time for heuristic methods for all datasets

10, $FilterRatio = 0.05$; Z-curve: $Shifts = 2$, $EntirePartitions = true$; LSH: W different for different datasets (20, 10, 200, 100), $HT = 5$, $HF = 20$. The results in Figure 30 show that for given settings the PAKJ approach has the highest precision, the Z-curve is the second best method on 200 and 1000-dimensional datasets and the LSH method displays the second best precision on the 10 and 512 datasets. In Figure 31, we can observe that despite having the highest precision, the PAKJ method is also the fastest algorithm on all measured datasets. The second fastest method is the Z-curve algorithm and LSH is the approach with the longest execution time.

5.5.2. Size-dependent computation

In Figures 32 and 33, we study the behavior of the compared methods for a significantly larger dataset. These tests run on the 200-dimensional dataset which consists of about 1.2 mil. database objects S and 15.3 mil. query objects R. With growing data size, we also increased the cluster size. Executors were set to {20, 40, 60, 80} and cluster size to {5, 10, 15, 20} instances (computing nodes) for objects count {4.1, 8.3, 12.4, 16.5} respectively. Despite our efforts to run experiments for all methods utilizing all objects (we even increased executor memory limits up to 4 GBs) not all methods were able to handle all test cases. The LSH algorithm run out of memory for the highest object count 16.5 mil. For the Z-curve method, we had to set the memory saving option to the value "Only Z-order" and $EntirePartitions = false$ to satisfy the memory limits. The results show that the PAKJ method provides the highest precision while keeping reasonably fast running time, the Z-curve with fast evaluation time provides poor precision and the LSH method is both slow and has low precision.

5.5.3. K-dependent computation

In the graphs 34 and 35, we investigate the influence of increasing parameter k (the number of nearest neighbors) on the precision and the similarity join

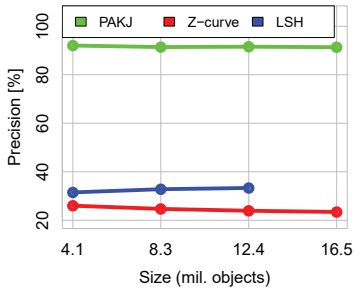

Figure 32: Growing dataset and cluster size - precision for all methods

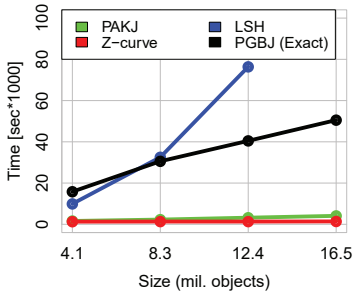

Figure 33: Growing dataset and cluster size - running time for all methods

evaluation time. All the presented experiments were performed on the 1000-dimensional dataset. Parameters were set as follows. PAKJ: $MaxRecDepth = 10$, $FilterRatio = 0.01$; Z-curve: $Shifts = 2$, $EntirePartitions = true$; LSH: $W = 50$, $HT = 10$, $HF = 20$. The precision slowly decreases for all methods, whereas the evaluation time is increasing for the PAKJ method, but for the other two methods the evaluation time (already high) is not changing significantly. The results of all the methods follow the trends identified in previous graphs. The pivot space approach outperforms other algorithms in the precision/speed trade-off.

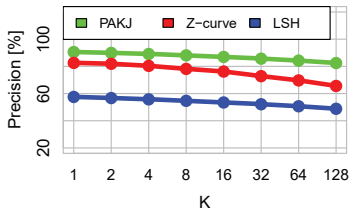

Figure 34: k-dependent computation - precision for all methods

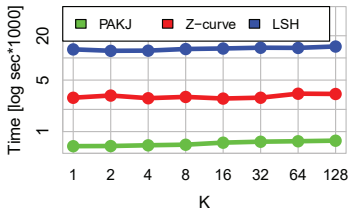

Figure 35: k-dependent computation - running time for all methods

5.6. Discussion

We analyzed the pivot-based approximate k-NN similarity join algorithms. We focused mainly on the approximation performance (precision, effective error, distance ratio) with respect to the number of replications and the execution time of all algorithms. The PEGKJ algorithm presents a guaranteed $\epsilon-$approximation but almost all objects are replicated to all groups on real data even with higher ϵ values, which results in longer execution times. With higher ϵ values, the approximation still provides a high precision for parameter $ExactPF = true$. The PAKJ algorithm runs faster and, in the average cases,

produces the results with high precision and low effective error. However, a few PAKJ queries violate the $\epsilon-$approximation significantly depending on a specific dataset.

We also compared the pivot-based approximate method PAKJ with other heuristic methods designed for vector spaces – Z-curve and LSH. All the methods use a random initialization of data partitioning structures. The pivot-based approach using the repetitive Voronoi partitioning outperformed the other two methods in the precision/efficiency tradeoff. Our hypothesis is that for high-dimensional data the Z-curve and LSH methods suffer from random shifts and hash functions that do not reflect data distributions. We verified this hypothesis on our synthetic 10-dimensional dataset (a dimensionality similar to the ones used in the original Z-curve and LSH papers). In this case, all three methods produced the expected behavior (aligned to the results 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 16 and W in Figure 18) do not provide a significant performance boost or increase running time greatly. On the other hand, the pivot-based approach PAKJ uses representatives from the data distribution and employs pairwise distances to determine data replication strategies. As demonstrated also by metric access methods for k-NN search [11, 47], the distance-based approach can be also directly used as a robust and intuitive method for MapReduce-based approximate k-NN similarity joins in high-dimensional spaces.

6. Conclusions

In this paper, we focused on approximate k-NN similarity joins in the MapReduce environment implemented mainly in Spark. We studied approximation quality and guarantees for pivot-based methods from the theoretical and experimental perspectives and presented two different pivot-based approximate k-NN similarity join algorithms. Although previous comparative studies have been proposed for other approximate methods, these studies focused mainly on data with less than one hundred dimensions. In this paper we also compared other heuristic algorithms reimplemented in Spark for high-dimensional data. According to our findings, data dimensionality significantly affects the way all the algorithms compare to each other, even after improving the previously proposed heuristic methods. In this paper, we also discussed the advantages, limitations and drawbacks of the proposed methods and implemented revisions.

Additional improvements to similarity k-NN joins could be achieved by implementing more sophisticated (but highly efficient) methods of space transformations and data partitioning (in this paper we focused on simple and fast random initialization methods). Furthermore, restrictions to specific properties of given distance measures could also bring improvements in the form of additional saved distance computations and faster execution times.

Acknowledgments

This project was supported by the Charles University in Prague grant GAUK 201515 and the Czech Science Foundation (GAČR) project Nr. 17-22224S.

References

[1] P. Cech, J. Marousek, J. Lokoc, Y. N. Silva, J. Starks, Comparing mapreduce-based k-nn similarity joins on hadoop for high-dimensional data, in: G. Cong, W. Peng, W. E. Zhang, C. Li, A. Sun (Eds.), Advanced Data Mining and Applications - 13th International Conference, ADMA 2017, Singapore, November 5-6, 2017, Proceedings, Vol. 10604 of Lecture Notes in Computer Science, Springer, 2017, pp. 63–75.

[2] P. Cech, J. Kohout, J. Lokoc, T. Komárek, J. Marousek, T. Pevný, Feature extraction and malware detection on large HTTPS data using mapreduce, in: L. Amsaleg, M. E. Houle, E. Schubert (Eds.), Similarity Search and Applications - 9th International Conference, SISAP 2016, Tokyo, Japan, October 24-26, 2016. Proceedings, Vol. 9939 of Lecture Notes in Computer Science, 2016, pp. 311–324.

[3] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, A. E. Abbadi, Approximate nearest neighbor searching in multimedia databases, in: Proceedings 17th International Conference on Data Engineering, 2001, pp. 503–511. doi:10.1109/ICDE.2001.914864.

[4] G. Giacinto, 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, CIVR '07, ACM, New York, NY, USA, 2007, pp. 456–463. doi:10.1145/1282280.1282347.

[5] C. Cobârzan, K. Schoeffmann, W. Bailer, W. Hürst, A. Blazek, J. Lokoc, S. Vrochidis, K. U. Barthel, L. Rossetto, Interactive video search tools: a detailed analysis of the video browser showdown 2015, Multimedia Tools Appl. 76 (4) (2017) 5539–5571. doi:10.1007/s11042-016-3661-2.
URL https://doi.org/10.1007/s11042-016-3661-2

[6] J. Lokoc, W. Bailer, K. Schoeffmann, B. Muenzer, G. Awad, On influential trends in interactive video retrieval: Video browser showdown 2015-2017, IEEE Transactions on Multimedia (2018) 1–1doi:10.1109/TMM.2018.2830110.

[7] G. R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Trans. Database Syst. 24 (2) (1999) 265–318. doi:10.1145/320248.320255.
URL http://doi.acm.org/10.1145/320248.320255

[8] M. Muja, D. G. Lowe, Scalable nearest neighbor algorithms for high dimensional data, IEEE Trans. Pattern Anal. Mach. Intell. 36 (11) (2014) 2227–2240. doi:10.1109/TPAMI.2014.2321376.
URL https://doi.org/10.1109/TPAMI.2014.2321376

[9] J. Lokoc, J. Kohout, P. Cech, T. Skopal, T. Pevný, k-nn classification of malware in HTTPS traffic using the metric space approach, in: M. Chau, G. A. Wang, H. Chen (Eds.), Intelligence and Security Informatics - 11th Pacific Asia Workshop, PAISI 2016, Auckland, New Zealand, April 19, 2016, Proceedings, Vol. 9650 of Lecture Notes in Computer Science, Springer, 2016, pp. 131–145.

[10] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.

[11] P. Zezula, G. Amato, V. Dohnal, M. Batko, Similarity Search - The Metric Space Approach, Vol. 32 of Advances in Database Systems, Kluwer, 2006. doi:10.1007/0-387-29151-2.
URL https://doi.org/10.1007/0-387-29151-2

[12] G. Song, J. Rochas, F. Huet, F. Magoulès, 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, 2015, pp. 279–287. doi:10.1109/PDP.2015.79.

[13] G. Song, J. Rochas, L. E. Beze, F. Huet, F. Magoulès, K nearest neighbour joins for big data on mapreduce: A theoretical and experimental analysis, IEEE Trans. Knowl. Data Eng. 28 (9) (2016) 2376–2392. doi:10.1109/TKDE.2016.2562627.
URL https://doi.org/10.1109/TKDE.2016.2562627

[14] M. Patella, P. Ciaccia, The many facets of approximate similarity search, in: Proceedings of the 24th International Conference on Data Engineering Workshops, ICDE 2008, April 7-12, 2008, Cancún, México, IEEE Computer Society, 2008, pp. 308–319. doi:10.1109/ICDEW.2008.4498340.
URL https://doi.org/10.1109/ICDEW.2008.4498340

[15] G. R. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, in: L. M. Haas, A. Tiwary (Eds.), SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA., ACM Press, 1998, pp. 237–248. doi:10.1145/276304.276326.
URL http://doi.acm.org/10.1145/276304.276326

[16] Y. N. Silva, J. M. Reed, Exploiting mapreduce-based similarity joins, in: K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, A. Fuxman (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, ACM, 2012, pp. 693–696. doi:10.1145/2213836.2213935.
URL http://doi.acm.org/10.1145/2213836.2213935

[17] Y. Ma, X. Meng, S. Wang, Parallel similarity joins on massive high-dimensional data using mapreduce, Concurrency and Computation: Practice and Experience 28 (1) (2016) 166–183. doi:10.1002/cpe.3663.
URL https://doi.org/10.1002/cpe.3663

[18] C. Böhm, F. Krebs, The k-nearest neighbour join: Turbo charging the KDD process, Knowl. Inf. Syst. 6 (6) (2004) 728–749.
URL http://www.springerlink.com/index/10.1007/s10115-003-0122-9

[19] W. Lu, Y. Shen, S. Chen, B. C. Ooi, Efficient processing of k nearest neighbor joins using mapreduce, Proc. VLDB Endow. 5 (10) (2012) 1016–1027. doi:10.14778/2336664.2336674.

[20] R. Vernica, M. J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: A. K. Elmagarmid, D. Agrawal (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, ACM, 2010, pp. 495–506. doi:10.1145/1807167.1807222.
URL http://doi.acm.org/10.1145/1807167.1807222

[21] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, X. Du, Fast and scalable distributed set similarity joins for big data analytics, in: 33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017, IEEE Computer Society, 2017, pp. 1059–1070. doi:10.1109/ICDE.2017.151.
URL https://doi.org/10.1109/ICDE.2017.151

[22] C. Xiao, W. Wang, X. Lin, Ed-join: an efficient algorithm for similarity joins with edit distance constraints, PVLDB 1 (1) (2008) 933–944.
URL http://www.vldb.org/pvldb/1/1453957.pdf

[23] J. Wang, G. Li, J. Feng, Trie-join: Efficient trie-based string similarity joins with edit-distance constraints, PVLDB 3 (1) (2010) 1219–1230.
URL http://www.comp.nus.edu.sg/ vldb2010/proceedings/files/papers/R108.pdf

[24] E. H. Jacox, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2) (2008) 7:1–7:38. doi:10.1145/1366102.1366104.
URL http://doi.acm.org/10.1145/1366102.1366104

[25] C. Böhm, B. Braunmüller, F. Krebs, H. Kriegel, Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data, in: S. Mehrotra, T. K. Sellis (Eds.), Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001, ACM, 2001, pp. 379–388. doi:10.1145/375663.375714.
URL http://doi.acm.org/10.1145/375663.375714

[26] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: L. Liu, A. Reuter, K. Whang, J. Zhang (Eds.), Proceedings of the 22nd International Conference on Data Engineering,

ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, IEEE Computer Society, 2006, p. 5. doi:10.1109/ICDE.2006.9.
URL https://doi.org/10.1109/ICDE.2006.9

[27] Y. N. Silva, S. S. Pearson, J. Chon, R. Roberts, Similarity joins: Their implementation and interactions with other database operators, Inf. Syst. 52 (2015) 149–162. doi:10.1016/j.is.2015.01.008.
URL https://doi.org/10.1016/j.is.2015.01.008

[28] W. Kim, Y. Kim, K. Shim, Parallel computation of k-nearest neighbor joins using mapreduce, in: J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, T. Suzumura (Eds.), 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016, IEEE, 2016, pp. 696–705. doi:10.1109/BigData.2016.7840662.
URL https://doi.org/10.1109/BigData.2016.7840662

[29] M. Tang, Y. Yu, W. G. Aref, Q. M. Malluhi, M. Ouzzani, Efficient processing of hamming-distance-based similarity-search queries over mapreduce, in: G. Alonso, F. Geerts, L. Popa, P. Barceló, J. Teubner, M. Ugarte, J. V. den Bussche, J. Paredaens (Eds.), Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015., OpenProceedings.org, 2015, pp. 361–372. doi:10.5441/002/edbt.2015.32.
URL https://doi.org/10.5441/002/edbt.2015.32

[30] C. Yu, B. Cui, S. Wang, J. Su, Efficient index-based KNN join processing for high-dimensional data, Information & Software Technology 49 (4) (2007) 332–344. doi:10.1016/j.infsof.2006.05.006.
URL https://doi.org/10.1016/j.infsof.2006.05.006

[31] Y. Ma, S. Jia, Y. Zhang, A novel approach for high-dimensional vector similarity join query, Concurrency and Computation: Practice and Experience 29 (5). doi:10.1002/cpe.3952.
URL https://doi.org/10.1002/cpe.3952

[32] Y. N. Silva, J. M. Reed, L. M. Tsosie, Mapreduce-based similarity join for metric spaces, in: Proceedings of the 1st International Workshop on Cloud Intelligence, Cloud-I '12, ACM, New York, NY, USA, 2012, pp. 3:1–3:8. doi:10.1145/2347673.2347676.
URL http://doi.acm.org/10.1145/2347673.2347676

[33] Y. Hu, C. Yang, C. Ji, Y. Xu, X. Li, Efficient snapshot KNN join processing for large data using mapreduce, in: 22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016, Wuhan, China, December 13-16, 2016, IEEE Computer Society, 2016, pp. 713–720. doi:10.1109/ICPADS.2016.0098.
URL https://doi.org/10.1109/ICPADS.2016.0098

[34] C. Zhang, F. Li, J. Jestes, Efficient parallel knn joins for large data in mapreduce, in: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, ACM, New York, NY, USA, 2012, pp. 38–49. doi:10.1145/2247596.2247602.

[35] A. Stupar, S. Michel, R. Schenkel, Rankreduce - processing k-nearest neighbor queries on top of mapreduce, in: LSDS-IR, 2010.

[36] P. Zhu, X. Zhan, W. Qiu, Efficient k-nearest neighbors search in high dimensions using mapreduce, in: 2015 IEEE Fifth International Conference on Big Data and Cloud Computing, 2015, pp. 23–30. doi:10.1109/BDCloud.2015.51.

[37] D. Moise, D. Shestakov, G. Þ. Gudmundsson, L. Amsaleg, Indexing and searching 100m images with map-reduce, in: International Conference on Multimedia Retrieval, ICMR'13, Dallas, TX, USA, April 16-19, 2013, 2013, pp. 17–24. doi:10.1145/2461466.2461470.

[38] D. Moise, D. Shestakov, G. Þ. Gudmundsson, L. Amsaleg, 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, 2013, pp. 674–682. doi:10.1109/BigData.2013.6691637.

[39] G. Þ. Guðmundsson, L. Amsaleg, B. Þ. Jónsson, M. J. Franklin, 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, 2017, pp. 1–12. doi:10.1145/3083187.3083200.

[40] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65. doi:10.1145/2934664.

[41] B. Yao, F. Li, P. Kumar, K nearest neighbor queries and knn-joins in large relational databases (almost) for free, ICDE.

[42] E. Schubert, A. Zimek, H. Kriegel, Fast and scalable outlier detection with approximate nearest neighbor ensembles, in: M. Renz, C. Shahabi, X. Zhou, M. A. Cheema (Eds.), Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part II, Vol. 9050 of Lecture Notes in Computer Science, Springer, 2015, pp. 19–36.

[43] F. Angiulli, C. Pizzuti, Outlier mining in large high-dimensional data sets, IEEE Trans. Knowl. Data Eng. 17 (2) (2005) 203–215. doi:10.1109/TKDE.2005.31.
URL https://doi.org/10.1109/TKDE.2005.31

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

[45] B. Bustos, G. Navarro, E. Chavez, Pivot selection techniques for proximity searching in metric spaces, Pattern Recognition Letters 24 (14) (2003) 2357 – 2366.

[46] E. Chávez, G. Navarro, R. Baeza-Yates, J. L. Marroquín, Searching in metric spaces, ACM Comput. Surv. 33 (3) (2001) 273–321. doi:10.1145/502807.502808.
URL http://doi.acm.org/10.1145/502807.502808

[47] D. Novak, M. Batko, Metric index: An efficient and scalable solution for similarity search, in: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications, IEEE, Washington, DC, USA, 2009, pp. 65–73. doi:10.1109/SISAP.2009.26.

[48] E. Chavez Gonzalez, K. Figueroa, G. Navarro, Effective proximity retrieval by ordering permutations, IEEE Trans. Pattern Anal. Mach. Intell. 30 (9) (2008) 1647–1658. doi:10.1109/TPAMI.2007.70815.

[49] P. Ciaccia, M. Patella, PAC nearest neighbor queries: Using the distance distribution for searching in high-dimensional metric spaces, in: E. Bertino, S. Castano (Eds.), Atti del Settimo Convegno Nazionale Sistemi Evoluti per Basi di Dati, SEBD 1999, Villa Olmo, Como, Italy, 23-25 Giugno 1999, 1999, pp. 259–273.

[50] P. Ciaccia, M. Patella, PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces, in: D. B. Lomet, G. Weikum (Eds.), Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000, IEEE Computer Society, 2000, pp. 244–255. doi:10.1109/ICDE.2000.839417.
URL https://doi.org/10.1109/ICDE.2000.839417

[51] J. Kohout, T. Pevný, Unsupervised detection of malware in persistent web traffic, in: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015, IEEE, 2015, pp. 1757–1761. doi:10.1109/ICASSP.2015.7178272.
URL https://doi.org/10.1109/ICASSP.2015.7178272

[52] J.-M. Marin, K. Mengersen, C. P. Robert, Bayesian modelling and inference on mixtures of distributions, in: D. Dey, C. Rao (Eds.), Bayesian ThinkingModeling and Computation, Vol. 25 of Handbook of Statistics, Elsevier, 2005, pp. 459 – 507.

[53] J. Kohout, T. Komárek, P. Cech, J. Bodnár, J. Lokoc, Learning communication patterns for malware discovery in https data, Expert Syst. Appl. 101 (2018) 129–142. doi:10.1016/j.eswa.2018.02.010.
URL https://doi.org/10.1016/j.eswa.2018.02.010

[54] G. Awad, J. Fiscus, M. Michel, D. Joy, W. Kraaij, A. F. Smeaton, G. Quénot, M. Eskevich, R. Aly, G. J. F. Jones, R. Ordelman, B. Huet, M. Larson, Trecvid 2016: Evaluating video search, video event detection, localization, and hyperlinking, in: Proceedings of TRECVID 2016, NIST, USA, 2016.

[55] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR abs/1409.1556. arXiv:1409.1556.
URL http://arxiv.org/abs/1409.1556