MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce

Thomas Seidl Sergej Fries Brigitte Boden

RWTH Aachen University, Germany
Data Management and Data Exploration Group
{seidl,fries,boden}@cs.rwth-aachen.de

Abstract: Data analytics gets faced with huge and tremendously increasing amounts of data for which MapReduce provides a very convenient and effective distributed programming model. Various algorithms already support massive data analysis on computer clusters but, in particular, distance-based similarity self-joins lack efficient solutions for large vector data sets though they are fundamental in many data mining tasks including clustering, near-duplicate detection or outlier analysis.

Our novel distance-based self-join algorithm for MapReduce, MR-DSJ, is based on grid partitioning and delivers correct, complete, and inherently duplicate-free results in a single iteration. Additionally we propose several filter techniques which reduce the runtime and communication of the MR-DSJ algorithm. Analytical and experimental evaluations demonstrate the superiority over other join algorithms for MapReduce.

1 Introduction

As an ongoing trend, tremendously increasing amounts of data are collected in real-world applications of life science, engineering, telecommunication, business transactions and many other domains. For the management and analysis of these data, many different techniques and algorithms have been developed, ranging from basic database operations to high-level data mining approaches like clustering, classification or the detection of outliers. Processing huge data sets with millions or billions of records on a single computer exceeds the computation capabilities of single computing nodes due to limitations of disk space and/or main memory. Thus, it is indispensable to develop distributed approaches that run on clusters of several computers in parallel [RU11].

An important group of database operations are *joins*. Similarity self-joins, which are a special type of joins, play an important role in data analysis: data cleaning [CGK06, RRS00], near duplicate detection [XWLY08, Mon00], document similarity analysis [BML10] and data mining tasks like density-based clustering like DBSCAN [EKSX96, BBBK00] inherently join the input data based on similarity relationships and, therefore, will draw high benefit from efficient and scalable implementations of similarity self-joins.

In this paper, we study the distributed computation of *distance-based similarity self-joins*. A distance-based join $R \bowtie_\varepsilon S = \{r \circ s \,|\, d(r.A, s.A) \leq \varepsilon\}$ returns all pairs of objects (r, s) whose distance in attribute A does not exceed a maximum dissimilarity threshold, ε, which is called the *range* or the *radius* of the similarity join. In our applications, the domain is a multidimensional data space $\mathbb{R}^{dim}$, and distance measures include the L_p norms like the Euclidean distance L_2, Manhattan distance L_1 or the Maximum distance L_∞.

For the development of distributed query processing algorithms, a variety of structured programming models exists. Aside classic parallel programming, the MapReduce model was proposed by Google [DG04], and its open-source implementation Hadoop found wide-spread attention and usage.

In this paper, we study the computation of distance-based self-joins for vector data using the MapReduce programming model. Our proposed grid-based approach combines the advantages of a very simple implementation and at the same time high efficiency, especially in low- to medium-dimensional domains that often occur in for example density-based clustering. However, it can also be applied to high-dimensional data by performing a dimensionality reduction first (dimensionality reduction techniques for MapReduce are implemented in the Mahout framework[1]). Overall, the main contributions of this paper are the following:

- We propose the MR-DSJ algorithm which efficiently computes the distance-based self-join on vector data using MapReduce avoiding duplicate distance computations.
- We introduce efficient filtering techniques based on mindist approximations for reducing communication and computation costs.
- We show the effectiveness of the developed approach by formal proofs and the efficiency by experiments on synthetic and real-world datasets.

The remainder of this paper is organized as follows: Section 2 describes related work. In Section 3 our join algorithm MR-DSJ is introduced. Section 4 presents the experimental evaluation of our technique. Theoretical analyses and ideas for future work are provided in Section 5, and Section 6 concludes the paper.

2 Related Work

The join operator is a fundamental database operator and is important for a large set of database queries. A general θ-join of two (or more) relations R, S is defined as $R \bowtie_\theta S = \sigma_\theta\,(R \times S) = \{r \circ s \,|\, \theta(r,s)\}$. Depending on the used predicate θ, different kinds of joins like equi-, spatial-, or distance-join are defined. As an alternative to our distance-based formalization, similarity functions $sim : U \times U \to \mathbb{R}_0^+$ for some object domain U can be used which indicate a high similarity of objects by high values. Prominent examples are the set intersection measure or the cosine similarity [VCL10, BML10].

The simplest solution for the computation of a join is a nested loop over both relations, which, however, has the disadvantage of quadratic complexities for computation and I/O. These problems have lead to the development of advanced approaches [Dat06] including

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

block nested loop, sort-merge, hash- or partition-based or index-based techniques which try to alleviate one of these or both problems.

Not every approach can, however, efficiently cope with similarity joins in multidimensional vector spaces. For example there is no natural sorting of points in a multidimensional space, such that sort-merge join techniques are probably not very appropriate solutions. The similarity join of very large data sets also has often to grapple with the problem of not indexed data. Due to the size, and - in case of distributed data storage - due to the distributed data location, the building of auxiliary index structures can be too expensive. The widespread solution for this problem is the usage of partition-based schemes which often can be performed in on-line fashion [ZJ03]. Appropriate data partitioning can lead to a significant efficiency gain of the join algorithm, which is achieved by pruning unnecessary distance calculation between partitions which are located too far away from each other. A prominent partitioning scheme is a (equal-sized) grid. While it does not require any data distribution information, it provides good results for not too skewed data and different approaches make use of it [BBKK01, PD96]. If additional information about the data is available, the partitioning can also address the data skewness problem [DNSS92].

In general, the parallelization of joins leads to a higher efficiency. In this work we investigate a solution for the similarity join in MapReduce. Let us briefly recall its programming model before we describe existing join approaches based on it. In MapReduce, the data is given as a list of records that are represented as (key, value) pairs. Basically, a MapReduce program consists of two phases: In the “Map” phase, the records are arbitrarily distributed to different computing nodes (called “mappers”) and each record is processed separately, independent of the other data items. The map phase then outputs intermediate (key,value) pairs. In the “Reduce” phase, records having the same key are grouped together and processed in the same computing node (“reducer”). Thus, the reducers combine information of different records having the same key and aggregate the intermediate results of the mappers. The results are stored back to the distributed file system. A simple example for a MapReduce program for the word count problem is given in [DG04].

Join processing using the MapReduce framework has already found high attention. [BPE+10] provides an overview of common join strategies in MapReduce. In [PPR+09], MapReduce is compared with parallel DBMS. Recent extensions of Hadoop including HadoopDB [ABPA+09], Hadoop++ [DQRJ+10] or PACT [ABE+10] also have a specific focus on join processing. However, the vast majority of existing work about parallel joins refers to equi-joins. Afrati and Ullmann [AU10] present optimization strategies for multi-way equi-joins, but they do not approach similarity joins. Broadcasting join strategies (e.g. [BPE+10]) rely on the assumption that the join partners significantly differ in their size ($|R| \ll |S|$), which does not apply for self-joins.

The field of similarity joins on MapReduce also gained a high attention in the last few years. The k-NN joins for Euclidian spaces were addressed in [LSCO12] and in [ZLJ12]. In the latter the author exploits the space-filling curves and transform the kNN joins into a sequence of one-dimensional range searches. In [LSCO12] a Voronoi based partitioning allows for an effective join. A technique for similarity joins in metric spaces was presented in [SRT12]. Both, [SRT12] and [LSCO12] are data partitioning approaches which require one or multiple runs on the data before the join algorithm can start. In this work we focus on similarity joins on vector data and exploit properties of vector spaces using a grid-based approach to obtain an efficient join. The vector space representation and the choice of equi-sized grids allows for a simple single-iteration algorithm which is particularly well suited for low- to medium-dimensional data. Though metric space joins are potentially able to handle very high-dimensional data, we believe that a simple but fast method as we propose in this work is an enrichment for MapReduce-based solutions for similarity joins.

Besides the joins for vector and metric spaces, there exist similarity join approaches that exploit characteristics of certain data types: Baraglia et al. [BML10] propose a two-step similarity self-join for textual documents based on inverted lists. Afrati et al. [ASM$^+$12] present an approach for similarity joins on data given as strings or sets. Similarity joins for set and multisets data on MapReduce is addressed in [VCL10] and [MF12]. Zhang et al. developed a spatial join algorithm for two-dimensional complex shape objects [ZHL$^+$09].

A general problem for parallelized similarity joins is the avoidance of duplicate results [ASM$^+$12]. E.g., in [ASM$^+$12] lexicographic orderings are used to solve this problem. In [BML10], the *reference tile method*, first introduced in [DS00], was used. In our work, we propose a simple yet very efficient technique for avoiding duplicates.

A very general approach that also avoids duplicates is the θ-join algorithm for MapReduce by Okcan and Riedewald [OR11]. The authors propose an effective randomized balancing strategy to distribute all possible result tuples (i.e. each pair of objects) across a given set of reducers. In the reduce phase, an arbitrary join algorithm is run in each reducer to compute the join of the data points that are assigned to this reducer. Though it shows an optimal load balancing, the original θ-join does not prune any distance computations during the run. As the general algorithm does not make any assumptions about the type of join, every pair of objects has to be processed by a reducer. The authors propose strategies to avoid some computations from the start for special join types, however no strategy for similarity joins is given. A further property of the θ-join algorithm is the dependence of its data replication factor on the cluster size. A higher number of reducers (i.e., computational nodes) leads to a higher replication of the data. In contrast to this approach, our technique makes use of pruning, and the replication of the data is independent from the cluster size.

3 Efficient Distance Self-Join

In this section, we present MR-DSJ, our algorithm for a similarity self-join of vector data in MapReduce. We first introduce a basic approach showing the idea of the algorithm in Section 3.1 and prove its correctness in Section 3.2. In Section 3.3 we introduce improved techniques to reduce the number of distance computations and the communication overhead and provide an efficient implementation for MapReduce framework in Section 3.4. We give an analytical analysis of the basic approaches, which extends the experimental evaluation from Section 4 and analyzes the worst case scenario, possible bottlenecks and load balancing properties of the approach. We use the following notations for grid cells:

(1) $NC(c)$: Set of neighboring cells of cell c

(2) $cell(p)$: Cell containing point p

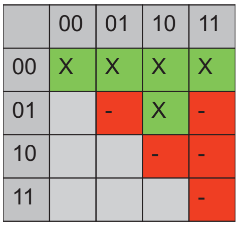


Figure 1: Bit Codes for the 2d case

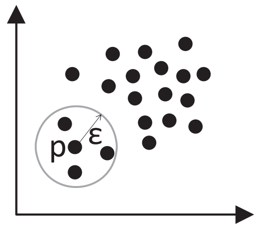


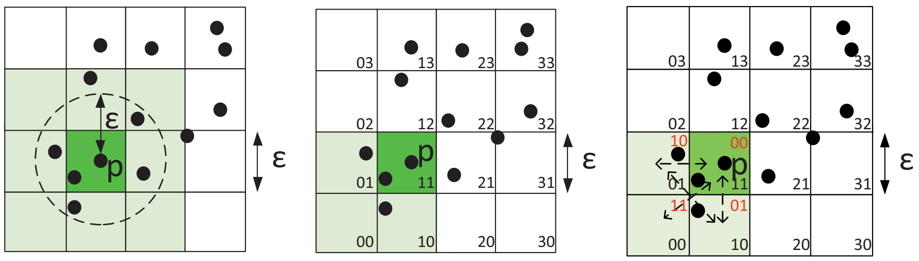
Figure 2: Small ε-neighborhood

3.1 MR-DSJ algorithm

A naive way for calculating joins is nested loop which computes the distance from each point to every other point. In the case of similarity joins where a result set can be very small (e.g. in the case of near-duplicate detection or clustering), this intuitive solution regularly produces far more distance calculations than necessary. For example, point p in Fig. 2 has only a small subset of database objects in its ε-neighborhood but nested loop would calculate the distances to all points. In order to reduce the number of distance computations we use a quite common grid-based partitioning approach with equal-sized grid cells of width ε. In such a grid all join partners of a point p are located either in the same cell as p or in the direct neighboring cells, and therefore all distance computations to objects in other cells can be pruned. This grid based discretization of the data space is depicted in Fig. 3(a) for Euclidean distance. It applies as well to other L_p norms, and weighted L_p norms are supported by scaled grid dimensions. For the point p lying in the dark green cell, each point in its ε-neighborhood is lying either in one of the adjacent (light-green) cells or in the cell of p itself. Using this knowledge, we can avoid the computation of the distances from p to the points in all other grid cells, because none of them would result in a valid result tuple.

This approach can be easily translated into a MapReduce program. Each reducer R_i is responsible for one cell c_i and its neighboring cells and computes - via nested loop - all the result tuples between all points located in c_i and $NC(c_i)$. We refer to c_i as the “home cell” of R_i. In the map phase, all points lying in a cell c_i are sent to the reducer R_i that is responsible for c_i and to the reducers of all adjacent cells, i.e. to all R_j for $c_j \in NC(c_i)$. In the reduce phase, each reducer R_i gets as input all points from c_i and $NC(c_i)$ and calculates the distances between all points in c_i and the distances of all points from c_i to all points from the neighboring cells via a nested loop. Since for each data point the distances to all objects in its neighbor cells are computed in some reducer, this method is a correct similarity self-join implementation. If ε is small compared to the distribution of values in the dataset, this simple approach can reduce the number of computations significantly.

However, this approach suffers from high communication overhead which stems from the 3^d times replicated data, since each point has to be sent to each reducer responsible for a cell neighboring to $cell(p)$. This high replication can be significantly decreased by reducing the number of neighbor cells that are taken into account by a single reducer. Namely, it is enough if each reducer R_i only considers the neighbor cells of c_i that have a



(a) Grid with cell width ε (b) Reducing the replication factor. (c) Avoiding duplicate results

Figure 3: Example grids (dark green cell: home cell, arrows indicate point replication between cells)

smaller or equal ID in every dimension, as shown in Fig. 3(b). Then the reducer performs a join on all the points from these cells. This approach still computes all valid result pairs because the same is done for *each* of the cells. For example, consider the cell c_{21}, which is the direct right neighbor of $cell(p) = c_{11}$. Although the reducer R_{11} of cell c_{11} does not compute the distance from p to the points from c_{21}, there exists another reducer R_{21} that has c_{21} as its home cell. Following the aforementioned rule, this reducer will also receive the points from c_{11} and then compute the distances between these two cells. Since the number of neighboring cells with smaller or equal ID in each dimension is equal to 2^d, this method replicates the data 2^d times, which is significantly smaller than 3^d.

Both, the 3^d and 2^d approaches though still suffer from the problem that lot of result pairs are duplicated, which occurs when two objects p and q from neighboring cells are processed in two separate reducers. This is e.g. the case when reducers of cells c_{11} and c_{21} both calculate the distances between cells c_{11} and c_{10}.
To avoid the unnecessary computation of duplicate result pairs, a reducer has to differentiate between the points from the different cells that were sent to it. Therefore we introduce a “bit code” that is sent with each data point to the reducers and identifies the relative position of the point’s cell to the home cell of the reducer. The bit codes consist of d bits (for a d-dimensional grid), where each bit corresponds to one dimension. The points of the home cell itself are assigned the bit code ‘0^d’$= 00\ldots0$ (d times). For the other cells, each bit indicates if the position of this cell deviates from that of the home cell in the corresponding dimension. An example is shown in Fig. 3(c), where the bit codes for the cells that are sent to the reducer R_{11} are presented. For example, the lower left cell is assigned the bit code ‘11’ because it differs from the home cell in both dimensions. Using these bit codes we can now decide, considering any particular reducer, which distances we have to compute in this reducer and which ones can be skipped as they are computed in other reducers. In Fig. 1 we exemplary show the decision matrices for the 2-dimensional case. For each pair of cells (represented by their bit codes) an ‘$\times$’ indicates that the distances between the points from this cells have to be computed in the considered reducer, while a ‘-’ indicates that the computations can be skipped as they are done in another reducer. The lower half of the matrix can be skipped due to symmetry. As a first rule, we only have to compute the distances between points from the same cell if it is the home cell of the considered reducer, because each of the other cells is the home cell of another reducer, thus

the distances between its points will be computed there. As another rule, we compute all the distances from the points in the home cell to the points in other cells. As a next step we determine all the distance computations between cells that will already be done in another reducer. Intuitively, we skip all distance computations between cells that both differ from the home cell in the same dimension, i.e. both of their bit codes contain a '1' at the same position (we refer to this rule as to *1s-rule* further in the text). In this case we know that in some other reducer, the same two cells will be processed together again and will then both contain a '0' at this position, thus the distances will be computed there. Thus, for the 2-dimensional case in Fig. 3(c), we just have to compute the distances between the cells with the bit codes '01' and '10' besides the distances including points from the home cell. A formal proof for the correctness of this step will be given in Section 3.2.

Using this approach, our join algorithm is guaranteed not to produce any duplicate result pairs. This does not only save unnecessary distance computations, but also the need to eliminate duplicates after the join.

Analysis

Now we give an analysis for uniformly distributed data in a d-dimensional space in terms of (1) number of computations per reduce-job ($comp_{red}$), (2) input size/communication per reduce-job ($input_{red}$), (3) memory footprint per reduce-job (mem_{red}), (4) overall number of computations ($comp_{overall}$) and (5) overall communication of the algorithm ($comm_{overall}$). We assume that the attribute domains are [0; 1] that means that the number of the set of all cells in the grid is equal to ε^{-d} (ε = width of a grid cell) and each cell in the grid contains $C = \varepsilon^d \cdot |DB|$ objects. Due to the 1s-rule and the resulting dependencies each reducer of MR-DSJ has to store all objects from the assigned home cell c_{home} and all neighboring cells $NC(c_{home})$. Since each cell contains C objects, the input of a single reducer (which is also the communication of a single reducer) $input_{red}$ and the memory footprint of a single reducer mem_{red} are equal to $2^d \cdot C$. Please consider that for small ε values, the value of $2^d \cdot C$ decreases very fast with growing dimensionality d. Using the bit code information the memory consumption can be halved to $2^{d-1} \cdot C$. A detailed description of this reduction technique is presented in Section 3.4.
The overall communication $comm_{overall}$ of the job is equal to $\varepsilon^{-d} \cdot input_{red} = 2^d \cdot |DB|$. Further, each reducer performs $\frac{3^d+1}{2} \cdot C^2$ computations, such that the overall computations $comp_{overall}$ is equal to $\frac{3^d+1}{2} \cdot \varepsilon^{2d} \cdot |DB|^2$. Intuitively, the points of a single cell c_i are compared to a half of all neighboring cells (there are $\frac{3^d-1}{2}$ of them) and with c_i itself, that is, c_i is compared to $\frac{3^d-1}{2} + 1 = \frac{3^d+1}{2}$ cells.

The presented analysis only holds for uniformly distributed data. Now we consider the worst case for the algorithm which occurs when all objects of the dataset are concentrated in a single cell only. In this case each reducer around the cell $cell(p)$ and the reducer of this cell itself receives all objects of the database, i.e., the overall communication $comm_{overall}$ is $2^d \cdot |DB|$. Additionally each of these reducers has to store the complete database locally, such that $input_{red}$ and mem_{red} grow to $|DB|$. The overall number of computations $comp_{overall}$ is then equal to $\frac{|DB|^2}{2}$.

3.2 Effectiveness of the MR-DSJ algorithm

In this section we show the correctness, completeness and minimality of the MR-DSJ algorithm. Therefore we prove the following lemmata that prepare Theorem 1, which states the desired properties.

Let $R \bowtie_\varepsilon R = \{(id_p, id_q) \in R \times R \,|\, d(data_p, data_q) \leq \varepsilon\}$ be the desired similarity self-join result, and $out_{DSJ} \subseteq R \times R$ denote the set of tuples reported by the MR-DSJ algorithm. id_p denotes the ID of a point p and $data_p$ represents the object coordinates. For $(p, q) \in R \times R$, let s_p^i, s_p^i+1, s_q^i, s_q^i+1 be the slices in dimension i to which DSJ_map assigns $data_p$ and $data_q$, respectively. Furthermore, let b_p^i, b_q^i denote the bit codes of p, q in a slice of dimension i where p, q are present.

Lemma 1 (Completeness of DSJ_map) *For each pair $(p, q) \in R \bowtie_\varepsilon R$, there is a reducer which receives both partners p, q by the partitioning of DSJ_map.*

Proof 1 *Assume that the proposition is false, i.e., there is a pair $(p, q) \in R \bowtie_\varepsilon R$ which does not meet in any reducer. This only may happen if $s_p^i + 1 < s_q^i$ or $s_p^i > s_q^i + 1$ for at least one dimension i. As the slices have width ε, it follows that $data_p^i + \varepsilon < data_q^i$ or $data_p^i > data_q^i + \varepsilon$, respectively. This eventually implies $d(data_p, data_q) > \varepsilon$ which contradicts the assumption.* $\square$

Lemma 2 (COMPLETENESS AND MINIMALITY OF DSJ_REDUCE) *For each $(p, q) \in R \bowtie_\varepsilon R$, exactly one of the reducers performing DSJ_reduce emits (id_p, id_q).*

Proof 2 *For each dimension i, two objects $(p, q) \in R \bowtie_\varepsilon R$ are processed in exactly one slice of i since exactly the following cases may occur for their bit codes b_p^i, b_q^i:*

(i) $(b_p^i, b_q^i) = (1, 1)$*: The pair is not processed in this slice $s_p^i + 1 = s_q^i + 1$ but will be emitted by a reducer of the neighboring slice $s_p^i = s_q^i$ where $b_p^i = b_q^i = 0$ and case (iv) applies.*

(ii) $(b_p^i, b_q^i) = (1, 0)$*: The pair (p, q) is emitted by one of the reducers for this slice $s_p^i + 1 = s_q^i$ since q was not present in the preceding slice s_p^i, and p is not present in the subsequent slice $s_q^i + 1$.*

(iii) $(b_p^i, b_q^i) = (0, 1)$*: Symmetric case to (ii), the pair (p, q) is emitted in this slice $s_p^i = s_q^i + 1$.*

(iv) $(b_p^i, b_q^i) = (0, 0)$*: The pair (p, q) is emitted by a reducer for this slice $s_p^i = s_q^i$; it is not emitted in the neighboring slice $s_p^i + 1 = s_q^i + 1$ where the bits for dimension i are both set and case (i) applies. Neither in preceding slices $s < s_p^i$ nor in subsequent slices $s > s_p^i + 1$, the objects p or q are present.*

At all, the pair (p, q) is emitted by reducers of a single slice per dimension only. The intersection of these slices over all dimensions determines a single partition. As this partition is not empty, it is processed by exactly one reducer, and the proposition holds. $\square$

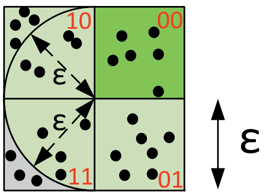


Figure 4: Pruning distance computations by $MindistCell$ (for the L_2 norm)

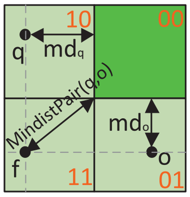


Figure 5: Example for MindistPair.

Lemma 3 (Correctness of DSJ_reduce) *MR-DSJ does not emit false positive pairs:* $out_{DSJ} \subseteq R \bowtie_\varepsilon R$.

Proof 3 *For each emitted reflexive pair* (id_n, id_n)*, the inequality* $d(data_n, data_n) = 0 \leq \varepsilon$ *trivially holds. Aside these, only pairs* (id_n, id_b) *and* (id_b, id_n) *are emitted for which* $d(data_p, data_q) \leq \varepsilon$ *was explicitly tested, and it holds that* $out_{DSJ} \subseteq R \bowtie_\varepsilon R$. □

Theorem 1 (Effectiveness of MR-DSJ) *The algorithm MR-DSJ produces complete and correct results without duplicates:* $out_{DSJ} = R \bowtie_\varepsilon R$.

Proof 4 *The completeness of MR-DSJ,* $out_{DSJ} \supseteq R \bowtie_\varepsilon R$*, follows from Lemmata 1 and 2, and the correctness of MR-DSJ,* $out_{DSJ} \subseteq R \bowtie_\varepsilon R$*, holds due to Lemma 3. The freedom of duplicates is equivalent to the minimality that any resulting pair is emitted by no more than a single reducer which was proven by Lemma 2.* □

3.3 Pruning distance computations

The basic solution provides a very efficient solution for grid-based similarity self-join on MapReduce, which decides whether a distance computation between two points is necessary by considering in which cells the points are contained. In this section we introduce two techniques to save even more distance computations and reduce replication by considering the position of points *within* a cell as will be shown in Sections 3.3.1 and 3.3.2.

3.3.1 Reducer side pruning by MindistCell

We start with an example in Fig. 4. The cell with bit code '11' contains some points that can in no case belong to a valid result pair including a point from the home cell, because their distance to *any* point in the home cell is greater than ε. For such points we do not have to compute the distances to all points from the home cell. A similar case occurs in the cell '10'. There exist some points such that none of them lies in the ε-neighborhood of *any* point from the cell '01', so we do also not have to compute the distances from those points to the points of the cell '01'. To exploit this facts for pruning distance computations, we first introduce the minimum L_p norm-based distance from a point to any point from a given cell, which is equal to the definition of the $MINDIST$ from [RKV95].

Definition 1 (*$MindistCell$*) *The distance of a point q to a cell c is defined as*

$$MindistCell(q,c) = \sqrt[p]{\sum_{i=1}^{d} \begin{cases} |lb_c[i] - q[i]|^p & q[i] < lb_c[i] \\ 0 & lb_c[i] \le q[i] \le ub_c[i] \\ |q[i] - ub_c[i]|^p & q[i] > ub_c[i] \end{cases}}, \text{ where}$$

$lb_c[i]$ denotes the lower bound and $ub_c[i]$ the upper bound of the cell c in dimension i.

In the reduce phase, we compute the $MindistCell$ of points q from each cell c_1 to each cell $c_2 \neq h_c$ such that the bit codes c_1 and c_2 differ in at least two dimensions. If $MindistCell(q, c_2) > \varepsilon$, the reducer does not compute the distance of q to *any* of the points in c_2.

Note that the $MindistCell$ pruning does not remove any relevant tuples such that the join result remains correct. For the reducer side pruning the proof is obvious, since for each object o and for each possible cell we test whether to prune o or not.

3.3.2 Mapper side pruning by MindistCell

The $MindistCell$ pruning is also applicable in the mapper which results in significant benefits. First, if a point is pruned in the mapper w.r.t. a certain cell c_i, it is not communicated to the reducer R_i such that the communication between mapper and reducer is reduced. This, secondly, induces that the replication factor of the overall approach decreases, which, in turn, additionally leads to a decreased runtime.

Technically, in the map phase, we detect for each home cell c_h the points q such that $MindistCell(q, c_h) > \varepsilon$. Please note that this is only possible for points from cells that differ from the home cell in at least two dimensions, thus we only have to compute the $MindistCell$ values for the points from those cells. For the mapper side pruning we show in theorem 2 that none of the pruned points occurs as a join partner in the reducer R_i. For that we briefly show which dimensions contribute to the $MindistCell(q, c)$ for a point q and a cell c in the Lemma 4.

Lemma 4 (Contribution of dimensions) *For a point q from cell c_q, and a cell c, only dimensions which are different in the bit codes for c_q and c contribute to $MindistCell(q, c)$.*

Proof 5 *According to definition of the bit code, if the values of the bit code are equal in a dimension i, then the range of the cells in i is equal. Therefore the middle rule of definition 1 applies and such a dimension does not contribute to $MindistCell(q, c)$.* □

Theorem 2 Completeness of map side $MindistCell$ pruning. *Let q be a map side pruned point, i.e., a point with $MindistCell(q, c_{home}) > \varepsilon$ then there is no other cell $c_j \in NC(c_{home})$ with $MindistCell(q, c_j) \le \varepsilon$.*

Proof 6 *Let $b^q = b_1^q, \cdots, b_d^q$ be the bit code of $cell(q)$ and $b^{c_j} = b_1^{c_j}, \cdots, b_d^{c_j}$ the bit code of the cell c_j. Additionally let I be the set $\{i|1 \le i \le d\}$ with $b_i^q = 1$. Then $|I|$ corresponds* *to the number of positions in which b^q differs from the bit code 0^d of the home cell and represents the dimensions which contribute to the $MindistCell$ according to Lemma 4. Lets now consider the cell c_j: due to the 1s-rule for every $b_i^{c_j}, i \in I$ must hold $b_i^{c_j} = 0$, since otherwise it will be pruned by the MR-DSJ algorithm. This in turn means that b^{c_j} has at least $(|I| + 1)$-many bits differing from b^q and therefore $MindistCell(q, c_j) > MindistCell(q, c_{home})$.* $\square$

3.3.3 Reducer side pruning by MindistPair

In the previous sections we described pruning techniques between a point and a cell. In this section we introduce *MindistPair* pruning between pairs of objects, which is defined in Definition 2.

Definition 2 *Let q, o be d-dimensional points in cells c_q, c_o, respectively, such that $bitcode(c_q)\&bitcode(c_o) = 0$, i.e., q and o are located in cells which are not pruned by the bit code pruning. Let c_{home} be the home cell, $md_q = MindistCell(q, c_{home})$ and $md_o = MindistCell(o, c_{home})$. Then $MindistPair(q, o)$ is defined as: $MindistPair(q, o) := \sqrt[p]{md_q^p + md_o^p}$, for $p \in \mathbb{N}$.*

Definition 2 states that given the $MindistCell$ to home cell for points q and o, the lower bound for the real distance between these points is L_p norm of their $MindistCell$s distances to the home cell.

Theorem 3 (Correctness of MindistPair pruning) *Let $q, o, c_q, c_o, c_{home}, md_q, md_o$ be defined as in Definition 2. Then it holds $MindistPair(q, o) \leq dist(q, o)$, where $dist(q, o)$ is a L_p or weighted L_p norm induced distance.*

At first we provide an intuitive explanation for this statement and then give a formal proof. Let points q and o in Fig. 5 be the two objects under consideration, the upper-right dark-green cell is the home cell and the arrows from q, o to the home cell represent the shortest distance to the home cell, i.e., the $MindistCell$s. The dashed lines represent the position of the point q on the x-axis and of the point o on the y-axis. Intuitively, the $MindistPair$ calculates the distance from the intersection of the dashed lines in point f to the home cell, which is always less or equal than the real distance between q and o. To be more precise, $MindistPair$ calculates the same distance for all points lying on the dashed lines in the cells c_q and c_o.

Proof 7 (MindistPair) *According to Definition 2, c_q and c_o differ in at most d dimensions and there is no dimension which contributes to md_q as well as to md_o at the same time. W.l.o.g. we assume that dimension i contributes to md_o with a value t_i, i.e., there is a hyperplane S of c_{home} to which the distance of o is t_i. If in the example in Fig. 5 i is the y-axis, then the hyperplane S would be the middle vertical line and t_i is the distance of o to this middle line. The distance $|q_i - o_i|$ is, however, equal to $t_i + x$, where $x \geq 0$ is the distance of q to the hyperplane S (in the example, x would the distance from the middle line to point q). I.e., $t_i \leq |q_i - o_i|$ for every dimension i. Therefore it holds that $\sqrt[p]{\sum_{i=1}^{d} t_i^p} \leq$*

$\sqrt[p]{\sum_{i=0}^{k} |q_i - o_i|^p}$. *If c_q and c_o differ in less than d dimensions, then according to Lemma 4, the dimensions which do not differ do not contribute to the $MindistCell$ and, therefore, the proof also holds for this case.*

3.4 Implementation of MR-DSJ algorithm

In the preceding sections, we have introduced the concepts of our MR-DSJ algorithm. Now we present its implementation, which consists of the Listings 1 and 2 for the mapper and the reducer, respectively. Both rely on a few global input parameters, namely the radius ε of the similarity join, the dimensionality of the data and the range of the data space, in terms of $bits\_per\_dimension$. Let us walk through our pseudocode. The recursion in the mapper (Listing 1) is started by the method *DSJ_map* which for each point first calculates the home slice and the distance to the upper slice boundary in each dimension. The value of the similarity radius ε is used to define the width of the slices. The recursion in *map_recursive* over the dimensions is initialized by a zero partition ID and a zero bit code. The parameter mdp aggregates the p-th power of the mindist from the point to the respective adjacent cells it is assigned to by summing up dist$[dim]^p$ over dimensions dim where the bit code is set to 1. That means that no contributions to mindist are added in dimensions of home slices, i.e., where the respective bit equals 0. The recursive calls for adjacent slices are conditional to the test if the mindist does not exceed ε (tested by $md^p \leq \varepsilon^p$). This way, the mapper-side mindistCell test is implemented with almost no additional effort compared to the basic variant. The value of mdp is handed over to the reducer for further mindist-based pruning.

The MR-DSJ reducer (Listing 2) cascades **for** and **if** statements to realize the respective loops and pruning strategies introduced in Section 3.3. Within the loop over the value records from the reducer's input, the second loop iterates over individual buffers for each neighboring cell. This separate buffer organization allows for efficient bit code pruning and *mindistCell* filtering of cells as a whole and, this way, prevent from unnecessary iterations over the contents in a pruned cell. Only for the remaining adjacent cells, all objects are tested by the last *mindistPair* filter before the final exact distance check from the join condition, and the resulting pairs are emitted.

As an additional optimization, we use the bit codes c_n not only to prevent from duplicate distance computations and duplicate results but also for minimizing the main memory footprint in the reducers. The tuple $(c_n, id_n, data_n)$ is buffered only if $c_n < maxcode$ holds since the bitwise AND test includes the most significant bits (MSB), and all pairs with set MSBs disqualify in particular. Reading the input in increasing bit code order, thus, enables to safely exclude all tuples with set MSB from the buffer; all their potential join partners got inserted into the buffer in earlier steps but no one will arrive later. The MSB threshold for the bit codes is precomputed in advance by $maxcode = 2^{dimension-1}$, which may be implemented by $maxcode = 1 << (dimension - 1)$.
The $c_n < maxcode$ test saves up to half the main memory consumption on average over all the reducers as the mapper assigns every object to up to two partitions per dimension.

Listing 1: The MR-DSJ mapper recursively assigns objects $(id, coord)$ to neighboring partitions pid. The bit codes c reflect the local neighborhood relationship for each dimension dim, and mdp aggregates the p-th power of the minimum distance to objects in the respective partition.

```
void DSJ_map(int id, float [] coord)
  for $dim = 1..dimension$ do
    home_slice[$dim$] = int(coord[$dim$] / $\varepsilon$);
    dist [$dim$] = (home_slice[$dim$]+1) $* \varepsilon -$ coord[$dim$];
  map_recursive (1, 0, 0, 0.0, id, coord);

void map_recursive(int $dim$, long pid, int c, float $mdp$, int id, float[] coord)
  if ($dim \leq dimension$)
    pid = (pid $<<$ $bits\_per\_dimension$) + home_slice[$dim$];
    map_recursive($dim$+1, pid, c<<1, $mdp$, id, coord);

    $mdp$ = $mdp$ + dist[$dim$]$^p$;
    if ($mdp \leq \varepsilon^p$)
      map_recursive($dim$+1, pid+1, (c<<1)|1, $mdp$, id, coord);
  else
    emit(pid, $(c, mdp, id, data)$);
```

Technically, the required sorting of the values in ascending bitcode order for each reducer is accomplished by the ‘secondary sort’ functionality of MapReduce. The key for the shuffle phase does not just comprise the partition ID, but includes the bit code in order to sort the input with respect to ($partition\_id$, $bitcode$) in lexicographic order.

4 Experiments

In this section we present an experimental evaluation of our new MR-DSJ approach. In Section 4.1 we evaluate the scalability of our approach on synthetic data and compare it to the θ-join approach from [OR11]. In [OR11] the author propose the usage of specialized join algorithms inside the reducer tasks. Therefore we implemented RSJ join [BKS93] which is very well suited for low-/medium-dimensional vector data.

In Section 4.2 we evaluate the efficiency gain of the optimizations presented in Section 3.3 compared to the basic MR-DSJ algorithm. In Section 4.3 we evaluate our approach on real-world data sets.

All the experiments were performed on a cluster running Hadoop 0.20.2 and consisting of 14 nodes with 8 cores each that are connected via a 1 Gbit network. Each of the nodes has 16 Gb RAM. For each experiment the number of the performed distance computations as well as the runtime is measured. Runs of the algorithms were aborted if they did not finish within 8 hours.

Listing 2: The MR-DSJ reducer computes the join results. The bit codes prevent from both, duplicate distance calculations and duplicate results from concurrent reducers while reducing the local main memory footprint as well. The minimum distances of objects to cells avoid several distance calculations.

```
void DSJ_reduce(long partition_id , Iterator values)
  cellBuffer . clear () ;
  forall ($(c_n, mdp_n, id_n, data_n)$ in values)
    for ($c_b$=0; $c_b < maxcode$; $c_b++$)
      if ($c_n$ & $c_b$ == 0)   // bitcode filter
        if (mindist($data_n$, $c_b$) $\leq \varepsilon$) // mindistCell filter
          forall ($(mdp_b, id_b, data_b)$ in cellBuffer[$c_b$])
            if ($mdp_n + mdp_b \leq \varepsilon^p$) // mindistPair filter
              if ($d(data_n, data_b) \leq \varepsilon$) // join condition
                emit($id_n, id_b$);
                emit($id_b, id_n$); // symmetric pair, if desired
    if ($c_n < maxcode$)   // relies on secondary sort
      cellBuffer [$c_n$].insert(($mdp_n, id_n, data_n$));
    if ($c_n$ == 0)
      emit($id_n, id_n$);   // reflexive pair, if desired
```

4.1 Scalability on synthetic data

In this section we evaluate our join approach on synthetic datasets of different sizes and dimensionalities. In all our synthetic datasets, the attribute values are uniform distributed between 0 and 1.

In our first experiment, the database sizes of our synthetic datasets are varied from 1 million points to approximately 10 million points. All used datasets are 2-dimensional and are processed with the parameter $\varepsilon = 0.05$. The results are shown in Fig. 6 (number of calculations) and Fig. 7 (runtime). Please note the logarithmic scale on the y-axes. As expected, the runtimes and the number of performed distance calculations of both algorithms increase for increasing database sizes. For these 2-dimensional datasets, the number of distance calculations performed by MR-DSJ is by a factor 2 to 3 higher than that of θ-join (denoted by "TJ" in the figures), as the RSJ-join in the reduce phase of TJ saves many calculations. However, for all datasets, the runtimes of MR-DSJ are significantly lower (by a factor 3 to 5) than those of TJ. This difference can be explained by the fact that in TJ *each* pair of data points is processed by a common reducer. Even if the distance computations for many pairs can be pruned by the RSJ join in the corresponding reducers, the distribution of the data and the building of the internal indexes for RSJ leads to high runtimes. In MR-DSJ, however, only pairs of points that have a certain proximity are processed by a common reducer.

In the next experiment, we vary the dimensionality of our datasets from 2 to 4 dimensions. All datasets consist of ca. 1 million points and are processed with the parameter $\varepsilon = 0.05$. The results presented in Fig. 9 show that the runtimes for both algorithms increase with higher dimensionality, while the runtimes of the MR-DSJ approach are always

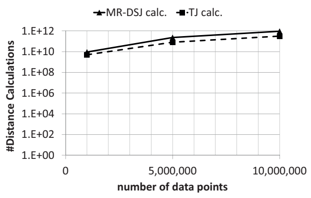


Figure 6: Database size vs. number of Distance Calculations

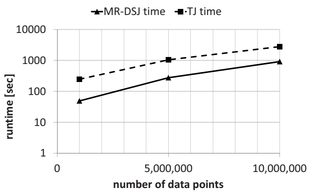


Figure 7: Database size vs. Runtime

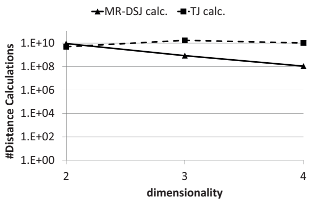


Figure 8: Dimensionality vs. number of Calculations

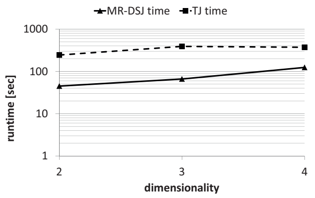


Figure 9: Dimensionality vs. Runtime

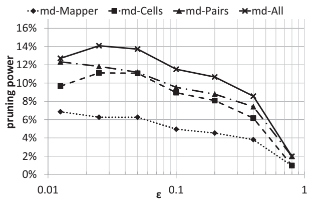


Figure 10: Efficiency gain by the optimizations on a 2d dataset

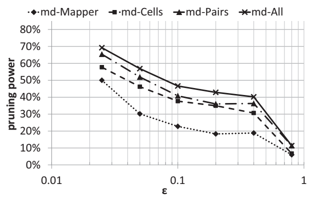


Figure 11: Efficiency gain by the optimizations on a 4d dataset

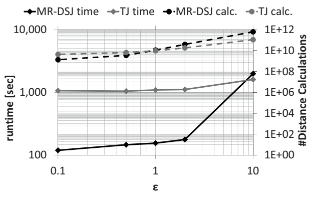


Figure 12: Varying ε on the cloud dataset

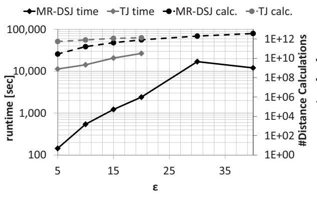


Figure 13: Varying ε on the minutiae dataset

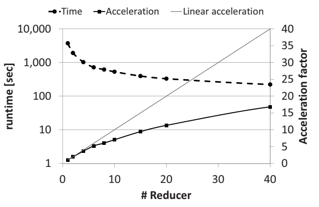


Figure 14: Scalability on the minutiae dataset

significantly lower than those of TJ. For the θ-join, the number of distance computations shown in Fig. 8 is hardly influenced by the dimensionality, as the partitioning strategy of this algorithm does not depend on the data dimensionality. In contrast, the number of distance computations for MR-DSJ strongly decreases for higher dimensionalities. This is caused by the fact that the data points are distributed among a larger number of grid cells for higher dimensionalities and thus for each point, the number of points in the same cell and the neighboring cells decreases, such that fewer distance computations have to be performed. The increasing runtime for higher dimensionalities results from the higher communication overhead between mappers and reducers which is caused by the higher replication factor.

4.2 Efficiency gain by the $MINDIST$ filters

In this section we evaluate the efficiency gain by the optimizations from Section 3.3. Therefore we use two synthetic datasets, with 2 and 4 dimensions. The experiment is repeated for different values of ε. For both datasets we measure the influence of each single filter and their combination in terms of saved distance calculations. The results are depicted in Fig. 10 (for the 2-dimensional dataset) and 11 (for the 4-dimensional dataset).

First, we evaluate the efficiency gain by using the mapper side pruning by computing the MindistCell (cf. Section 3.3.2) (denoted by "md-Mapper"). Depending on the ε value, we save up to 7% of the distance calculations that would be performed by the basic MR-DSJ algorithm on the 2-dimensional dataset and even up to 50% on the 4-dimensional dataset. The pruning power of this optimization, as well as the other optimizations, is the best for small ε values, which would be reasonable values for e.g. clustering or outlier detection. For ε values approaching 1 (please note that the synthetic data points lie in $[0, 1]$ in each dimension), naturally only a small amount of distance calculations can be pruned as the join selectivity approaches 77% (2-dimensional) and 53% (4-dimensional) and thus most distances have to be calculated. Additionally using the reducer side pruning by MindistCell (cf. 3.3.1) (denoted by "md-Cells"), leads to the pruning of up to 10% of the distance calculations for the 2-dimensional dataset and up to 58% for the 4-dimensional dataset. Using the reducer side pruning by MindistPair from Section 3.3.3 (together with the mapper side pruning), denoted by "md-Pairs", we can prune up to 12% of the distance calculations for the 2-dimensional dataset and up to 65% for the 4-dimensional dataset. Finally, the advanced MR-DSJ algorithm using all optimizations needs to perform up to 14% less distance calculations than the basic MR-DSJ algorithm for the 2-dimensional dataset and up to 70% less for the 4-dimensional dataset.

Overall, we observe that for the 4-dimensional dataset, the pruning power of the optimizations is much higher than for the 2-dimensional dataset. (This effect can also be observed in Fig. 8.) This can be explained by the fact that in higher-dimensional spaces, a larger percentage of the data points lie near the borders of their respective cell. As the optimizations mostly prune distance calculations for points near the borders, they are much more effective in a 4-dimensional space than in a 2-dimensional space.

4.3 Scalability on real world data

We evaluate our approach on two real-world datasets. The first dataset is a sample of 5 million records from a dataset of cloud observations from land stations and ships [HW99] that is available online[2]. We use a 2-dimensional dataset for which the attributes "latitude" and "longitude" are used. The second dataset "minutiae" contains extracted minutiae data from the fingerprint datasets "NIST Special Database 14" and "NIST Special Database 29[3]". It contains ca. 11 million three-dimensional entries, distributed in the ranges [0;832],

[2] `http://cdiac.ornl.gov/ftp/ndp026c/`

[3] `http://www.nist.gov/srd/nistsd{14,29}.cfm`

[0;768] and [0;100], respectively. As the efficiency of the MR-DSJ approach depends on the parameter ε which determines the size of the grid cells, we evaluated its scalability (and that of TJ) to different ε-values on both datasets. The results are shown in Fig. 12 and Fig. 13, respectively. Both experiments show the expected behavior for both algorithms - increasing ε values result in higher runtimes and number of performed calculations, which makes sense as for higher ε values we also get a larger result set.

The number of distance calculations as well as the runtime for different ε values vary significantly for the MR-DSJ approach. For $\varepsilon = 0.1$, MR-DSJ finishes on the cloud dataset (Fig. 12) in 119 seconds and performs ca. $1.3 \cdot 10^9$ distance calculations. For higher values for ε the number of needed calculations increases. For $\varepsilon = 10$ the runtime is approx. 2000 seconds and approx. $6 \cdot 10^{11}$ distance calculations are performed. Whereas the number of distance calculations of TJ behaves similar to that of MR-DSJ, the runtimes for MR-DSJ are significantly lower than those of TJ, except for the value $\varepsilon = 10$, which leads to very large grid cells in the MR-DSJ approach.

For the minutiae dataset (Fig. 13) our observations are similar the those for the cloud dataset. The runtimes and numbers of calculations increase for increasing ε values; for values larger than 20, the TJ approach did not finish within 8 hours and is thus not included in the figure. The numbers of performed distance calculations are again similar for both algorithms. However, MR-DSJ outperforms TJ in terms of runtime by a factor of 10 to 80.

In a further experiment (Fig. 14) we analyze the scalability of our approach on the minutiae w.r.t. a varying cluster size. Therefore we vary the number of used reducers from 1 to 40. In Fig. 14 we depict the runtimes of MR-DSJ as well as the acceleration factor compared to the runtime using only 1 reducer. For up to 6 reducers, the acceleration factor is nearly linear. For larger numbers of reducers, the acceleration is sub-linear which is mainly caused by data skew: As some grid cells contain more data points that other ones, the reduce tasks processing these cells have longer runtimes than those of the other reducers. However, MR-DSJ reaches a significant acceleration for increasing cluster sizes, i.e. using 40 reducers the runtime is lower than the runtime for 1 reducer by a factor of 17.

5 Further Analysis

In the previous sections we analyzed the basic MR-DSJ algorithm in terms of number of calculations. In this section we investigate its general behavior and present scenarios and use cases which benefit the most from its usage. We also identify problematic scenarios for MR-DSJ and present ideas how to tackle this problems in future work.

5.1 Data replication and data dimensionality

The effect of data replication is very common in the MapReduce framework. The only (desired) way to share information is its duplication on different computational nodes. Since each reducer in MR-DSJ relies on information from neighboring cells, the replication of data points is unavoidable. As shown earlier, our approach produces 2^d replicas of every data point, i.e. the replication factor grows very fast with every additional dimension of the data that is used for partitioning the data space. Thus, our approach is most suitable for data with a low or medium dimensionality d. Please note that d does *not* necessarily equal the dimensionality of the original data, since we can apply dimensionality reduction techniques (e.g. the techniques from the Mahout framework) to obtain a reasonably low dimensionality.

5.2 Influence of the parameter ε

In the analysis parts of Section 3 we already mentioned the worst case scenario for our approach: all points are located in a single grid cell and thus are processed by a single reducer. This case occurs for very skewed data or when the value of ε is very large in comparison to the largest distances between data objects. On the other hand, small ε values result in a high number of grid cells with few elements. In this case the computation of the self-join becomes very efficient since most of the distance computations can be pruned. Due to these facts our algorithm is best suited for join tasks with small ε values which among other things arise in near-duplicate detection, data cleaning and clustering tasks.

One possible way to solve the problem with large ε values would be a connection of our approach with an adjusted version of the θ-join [OR11]. The join inside cells containing too many objects would then be calculated by θ-join. This connection could also be helpful in the case of highly skewed data. We will examine such a connection in our future work.

The threshold ε also directly influences the number of created reduce tasks and therefore the possible parallelization of our approach. A small number of cells, which corresponds to a small number of created reduce-jobs, can significantly deteriorate the performance of the complete task. This case will for example occur if the chosen threshold ε is very large. A possible solution for this problem, which will be investigated in our future work, is adjusting the cell width to larger or smaller values than ε.

6 Conclusion

In this work we proposed the novel distance-based similarity self-join algorithm MR-DSJ for the MapReduce framework. We presented different optimization solutions for the used grid-based approach which minimize the communication and the number of needed distance computations. We provided a theoretical analysis of the used basic techniques as well as an experimental evaluation of the efficiency of our approach. The evaluation shows that our solution often significantly outperforms the existing θ-join algorithm in terms of number of calculations and execution runtime.

Acknowledgments This work has been supported by the B-IT Research School of the Bonn-Aachen International Center for Information Technology. We also thank Roman Haag for his technical support.

References

[ABE+10] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3:1625–1628, 2010.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[ASM+12] F.N. Afrati, A.D. Sarma, D. Menestrina, A. Parameswaran, and J.D. Ullman. Fuzzy Joins Using MapReduce. *ICDE*, 2012.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, pages 99–110, 2010.

[BBBK00] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pages 298–305, 2000.

[BBKK01] C. Böhm, B. Braunmüller, F. Krebs, and H.P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, pages 379–388, 2001.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, pages 237–246, 1993.

[BML10] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document Similarity Self-Join with MapReduce. In *ICDM*, pages 731–736, 2010.

[BPE+10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*, page 5, 2006.

[Dat06] Chris J. Date. *The relational database dictionary - a comprehensive glossary of relational terms and concepts, with illustrative examples*. O'Reilly, 2006.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, pages 27–40, 1992.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3:518–529, 2010.

[DS00] Jens-Peter Dittrich and Bernhard Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *SIGKDD*, pages 226–231, 1996.

[HW99] C.J. Hahn and S.G. Warren. Extended edited synoptic cloud reports from ships and land stations over the globe, 1952–1996. *NDP026C, Carbon Dioxide Information Analysis Center*, 1999.

[LSCO12] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. *PVLDB*, 5(10):1016–1027, 2012.

[MF12] Ahmed Metwally and Christos Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB*, 5(8):704–715, 2012.

[Mon00] Alvaro E. Monge. Matching Algorithms within a Duplicate Detection System. *IEEE Data Eng. Bull.*, 23(4):14–20, 2000.

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, pages 949–960, 2011.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD Conference*, pages 259–270, 1996.

[PPR$^+$09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest Neighbor Queries. In *SIGMOD*, pages 71–79, 1995.

[RRS00] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. In *SIGMOD*, pages 427–438, 2000.

[RU11] A. Rajaraman and J.D. Ullman. *Mining of massive datasets*. Cambridge Univ Pr, 2011.

[SRT12] Yasin N. Silva, Jason M. Reed, and Lisa M. Tsosie. MapReduce-based similarity join for metric spaces. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, Cloud-I '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM.

[VCL10] R. Vernica, M.J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, pages 495–506, 2010.

[XWLY08] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[ZHL$^+$09] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*, pages 1–8, 2009.

[ZJ03] Yanchang Zhao and Song Junde. AGRID: An Efficient Algorithm for Clustering Large High-Dimensional Datasets. In *PAKDD*, pages 271–282, 2003.

[ZLJ12] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel kNN joins for large data in MapReduce. In *EDBT*, pages 38–49, 2012.