Efficient Clustering of Massive Data with MapReduce

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Sergej Fries
aus Temirtau (Kasachstan)

Berichter: Universitätsprofessor Dr. rer. nat. Thomas Seidl
Universitätsprofessor Dr.-Ing. habil. Erhard Rahm

Tag der mündlichen Prüfung: 01.12.2014

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

To those who supported me during the past years.

First, I would like to thank my parents, my brother and of course my wife Steffi, for always being there for me from the beginning and supporting me all the time. I hope you will be proud of me.

A special thank is to my supervisor Prof. Thomas Seidl, whose ideas and visions were the beginning of my research topic and who made this work possible at all. And heartfelt thanks for his support in all these years.

The years of writing wouldn't be that pleasant if there weren't nice colleagues and friends (Ines, Ina, Brigitte, Hardy and all the others!). Thanks to you guys for this great time!

Contents

Abstract / Zusammenfassung **1**

1 Overview **5**
1.1 KDD process 6
1.2 Clustering 9
1.3 MapReduce 11
1.4 Contributions and structure of the thesis 13
1.5 General remarks 14

I Density-based clustering of massive data **15**

2 Introduction to DBSCAN and basic idea **19**
2.1 Sequential DBSCAN algorithm 19
2.2 Basic idea of JBMR-DBSCAN 24

3 Similarity self-join of vector data **29**
3.1 Introduction 31
3.2 Related Work 33
3.3 MR-DSJ: Efficient Distance Self-Join of low-dimensional data 35
3.4 Experiments 50
3.5 Further Analysis 56
3.6 PHiDJ: Extending to high-dimensional spaces and skewed data 57
3.7 Experiments 87
3.8 Conclusion 98
3.9 Future work 98

4 CC-MR: Finding connected components in graphs 101
4.1 Introduction 102
4.2 Fundamentals 103
4.3 Related Work 104
4.4 Algorithm 107
4.5 Experiments 122
4.6 Conclusion 129

5 DBSCAN in MapReduce 131
5.1 Related work 132
5.2 JBMR-DBSCAN approach 136
5.3 Integrated JBMR-DBSCAN approach 139
5.4 Evaluation 150
5.5 Summary and conclusion 157
5.6 Future work 158

II Projected clustering on massive data 159

6 MR-P3C 163
6.1 Introduction 164
6.2 Related Work 167
6.3 Original P3C algorithm 169
6.4 Boosting P3C model for Big Data 175
6.5 P3C$^+$-MR in MapReduce 182
6.6 P3C$^+$-MR-Light algorithm 189
6.7 Experiments 192
6.8 Conclusion 202
6.9 Future work: Steering the P3C$^+$ algorithm 203

7 Discussion: Further approaches 205

III Summary and Outlook 209

IV Appendices I

List of Algorithms and Listings III

Bibliography V

Statement of Originality XXI

List of Publications XXIII

Abstract

In God we trust, all other bring data.
– William Edwards Deming (1900-1993)

Since several decades, after the Agrarian society and Machine Age, the mankind approached the Information Age. Information or even much more important knowledge became one of the most valuable resources. The usual way to generate knowledge is the analysis of observation, or of some raw data, and the more and interconnected data is available the more insights can be gained from it. Therefore, in the past decade the trend to gather all possible information in all areas of life, industry and science became overwhelming. Moreover, the technological development of storage and sensor systems allowed an even larger growth of data that are stored. As stated by Peter Hirshberg (global pulse summit) the amount of generated data in the year 2011 alone has exceeded the amount of data generated since the beginning of mankind's history. The importance of knowledge extraction led to the development of the Knowledge Discovery process in Databases (KDD process) in the year 1996. The KDD process describes a workflow from the raw data gathering, its preprocessing, and analysis to the final visualization for further interpretation.

In the last decades, the model-driven approach for knowledge extraction was mainly used. That is, the gathered data was used to accept or to decline a hypothesis that was developed by a human expert. Therefore, the accuracy of the predictive quality of the model highly depended on the expertise of the specialist. Moreover, even good models could miss several aspects of the problem at hand. In the last years, the data-driven approach for knowledge

extraction gained a lot of attention. The idea is letting the data “speak for themselves”, i.e., to generate novel models based on the given data and to validate them afterwards. As the models are not known in advance, the goal is to find unknown patterns in the data. In the KDD process, this task is usually solved by a group of data mining techniques called *unsupervised learning* or *cluster analysis*.

However, the cluster analysis task is often computationally expensive and efficient techniques for huge amount of data are indispensable. The usual way for processing large amounts of data is the parallelization of single tasks on multi-core or in cluster environments. In this work, the author follows the parallelization approach and investigates and presents novel techniques for processing and analyzing huge datasets in the widely used MapReduce framework. MapReduce is a parallelization framework for data intensive task that was proposed by Google Inc. in 2004 and developed to one of the most prevalent technologies for batch processing of huge amounts of data.

More precisely, this thesis deals with two classes of cluster analysis – the density-based approaches and particularly DBSCAN algorithm, and the projected clustering techniques, where the P3C algorithm was investigated and further developed for processing huge datasets. As part of the density-based approaches, the author of this thesis proposes efficient approaches for similarity self-join technique in vector spaces and determination of connected components in huge graphs in the MapReduce framework.

Zusammenfassung

In God we trust, all other bring data.
– William Edwards Deming (1900-1993)

Nach der Epoche der Agrargesellschaft und dem Industriezeitalter befindet sich die Menschheit seit einigen Jahrzehnten im Informationszeitalter. Die Information oder was viel wichtiger ist das Wissen wurde zur kostbarsten Ressource. Die bisherigen Ansätze zur Generierung des Wissens ist die Analyse von Beobachtungen oder anderen Rohdaten. Und mit wachsender Vernetzung der Daten wächst auch der Anteil des Wissens, das daraus extrahiert wird. Aus diesem Grund war der Trend des letzten Jahrzehnts die Sammlung aller möglichen Informationen in allen Bereichen unseres Lebens, sei es in der Industrie, der Wissenschaft oder allgemein der Gesellschaft. Die technologische Entwicklung der Speichermedien und der Sensortechnologien führten zum immer größer werdenden Zuwachs an Daten. So wurde von Peter Hirshberg (global pulse summit) vorhergesagt, dass allein im Jahr 2011 mehr Daten generiert würden als während der gesamten Geschichte der Menschheit vorher. Die Wichtigkeit der Wissensextraktion führte zur Entwicklung des Wissensextraktionsprozesses in Datenbanken (Knowledge Discovery process in Databases) im Jahre 1996. Der KDD-Prozess beschreibt einen Workflow von rohen Daten, über ihre Vorverarbeitung, Analyse bis zur endgültigen Visualisierung für weitere Interpretationen.

In den letzten Jahrzehnten dominierte bei der Wissensextraktion das modellgeleitete Vorgehen. D.h., die gesammelten Daten wurden zur Annahme oder zur Ablehnung eines von einem menschlichen Experten erstellten Modells verwendet. Somit hing die Vorhersagegenauigkeit des Modells sehr stark

von der Expertise der Spezialisten ab. Sogar gute Modelle berücksichtigen unter Umständen nicht alle Aspekte des gegebenen Problems. In den letzten Jahren rückten datengeleitete Ansätze zur Wissensextraktion mehr in den Vordergrund. Die Idee dabei ist, die Daten "für sich sprechen zu lassen", d.h. die Modelle direkt aus den Daten zu generieren und diese erst danach zu validieren. Da die Modelle im Vorhinein nicht bekannt sind, ist das Hauptziel, unbekannte Muster in den Daten zu finden. Im KDD Prozess wird diese Aufgabe von sogenannten Clusteranalyse-Techniken des Data Minings gelöst.

Die Verfahren der Clusteranalyse haben jedoch oft eine hohe Rechenkomplexität, so dass effiziente Methoden zur Analyse von großen Datenmengen unabdingbar sind. Ein weit verbreiteter Weg dazu ist die Parallelisierung von Algorithmen, die dann auf Multi-Core Computern oder auf Clustern ausgeführt werden können. Der Autor dieser Arbeit verfolgt den gleichen Lösungsweg und stellt neue Techniken zur Analyse von großen Datenmengen im MapReduce-Framework vor. MapReduce ist ein Parallelisierungsframework für datenintensive Anwendungen, das im Jahre 2004 von Google vorgestellt wurde. Seitdem entwickelte sich MapReduce zu einer der wichtigsten Technologien zur Verarbeitung von datenintensiven Aufgaben.

Den Schwerpunkt dieser Arbeit stellen zwei Klassen der Clusteranalyse dar, die dichtebasierten und Projected Clustering Verfahren. Dabei werden für jeweils einen Kandidaten aus jeder Klasse – und zwar den DBSCAN- und den P3C-Algorithmus – effiziente Verfahren vorgestellt und evaluiert. Als Teil der dichtebasierten Verfahren wurden in dieser Arbeit des Weiteren effiziente Techniken zum Selbstähnlichkeits-Join in Vektorräumen und ein Verfahren zur Bestimmung der Zusammenhangskomponenten in Graphen vorgestellt.

Chapter 1

Overview

Contents

1.1	KDD process	**6**
1.2	Clustering	**9**
1.3	MapReduce	**11**
1.4	Contributions and structure of the thesis	**13**
1.5	General remarks	**14**

From the beginning of digital data production, large data volumes are a problem, a challenge and an opportunity at the same time. The hidden patterns in the gathered data are often a source for scientific, social and economic interest. As more data is available the more clear and precise patterns can be extracted from it. Therefore, the analysis of existing data as for example experimental data (just think of CERN), data generated by telecommunication companies, or other industries are a great opportunity to discover new knowledge and allows for progress of the whole mankind.

However, extraction of the knowledge from the data is a complex task, which becomes more and more complicated the more data is available. Machine learning and data mining research provided great tools for data analysis in the last decades but many existing approaches require a large number of computations and I/O operations. This thesis investigates the question how to process huge amounts of data in reasonable time by parallelizing

existing techniques with the MapReduce framework.

MapReduce is a framework and a functional programming model that is widely used in many companies including Facebook, Yahoo and Google. It combines such properties as fault tolerance, parallelized I/O, easy to understand programming model and a technical simplicity, which lead to the great spread of this framework. Section 1.3 deals with MapReduce and related approaches.

In this work, we address one class of data mining algorithms called *clustering*. The goal of clustering techniques is a meaningful grouping of objects of a database that is usually used as a preprocessing step for other algorithm or as a standalone step for data analysis. The thesis deals with two classes of clustering techniques: The first one is density-based clustering (Part I) and the second one is projected clustering (Part II). A short introduction to clustering techniques is given in Chapter 1.2, and detailed descriptions are provided in the appropriate chapters. Before we proceed to the above-mentioned topics, a short introduction to the Knowledge Discovery Process and the context of this thesis will be given in Section 1.1.

1.1 KDD process

Knowledge Discovery in Databases process or in short *KDD process*, as defined by Fayyad et al.[FPSS96], refers to the overall process of discovering useful information from data. It consists of data management and preprocessing steps, data preparation, selection and cleaning as well as of the actual analysis step (data mining) and final visualization. Those steps are depicted in Figure 1.1. This means, the task of the KDD process is to transform the initial raw data into a human understandable knowledge. Each of these steps gained a lot of attention in the scientific communities and industry. In this thesis, we concentrate on the data mining part of the overall process.

Data mining is the actual analysis step of the KDD process and is usually associated with the application of different techniques on the data for discovering unknown patterns from the data. These techniques are usually

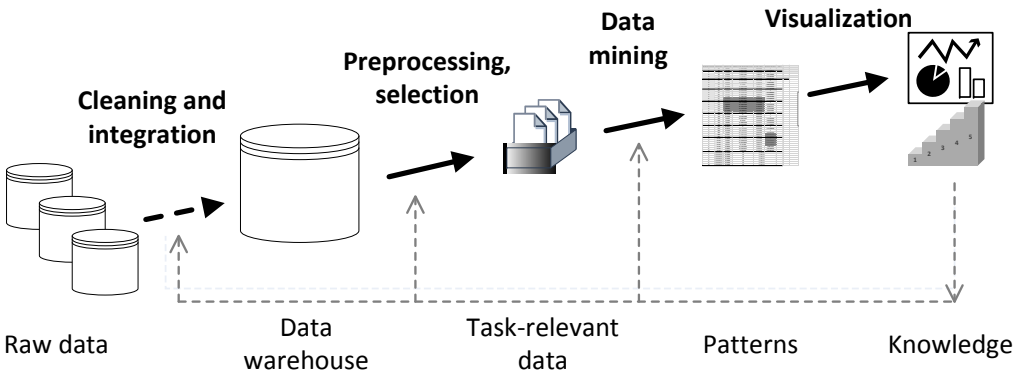


Figure 1.1: Steps of the KDD process.

subdivided into four areas: *supervised* learning or *classification/regression*, *unsupervised* learning or *clustering analysis*, *association rules mining* and *outlier mining*. In the following, we give a brief explanation to the first, and the last two areas and later on focus the attention on the unsupervised learning, as this is the main topic of this work.

Supervised learning The *supervised* learning addresses the problem of predicting of unknown classes or values of objects based on a classified training object sets. The variety of applications where the supervised learning techniques are employed is huge. Speaker and object recognition, tumor classification, customer type identification are only a very few examples where the classification and regression techniques are used. Often, the supervised learning techniques assume the existence of a parameterized *model*, those parameters are learned in the *training phase*. Given a trained model M, the classes, in case of classification, or continuous values in regression techniques can be predicted by applying M on new unclassified objects. In the last decades, several algorithm for supervised learning were proposed. They include among others *Support Vector Machines*, *Decision trees*, *Nearest Neighbor classifier*, *Neural Networks* and many more. An overview over supervised learning techniques can be found in [Kot07, Bis06].

Association rule mining The *association rule mining* is a method for discovering relations between variables in the databases in the form of rules $\{A_1, \ldots, A_k\} \Rightarrow \{B_1, \ldots, B_l\}$, where A_i and B_j are atomic items of the database. The most prominent example of the association rule mining is probably the discovering of regularities in the transactional supermarket databases. The result of the analysis is a set of rules where each rule $\{A_1, \ldots, A_k\} \Rightarrow \{B_1, \ldots, B_l\}$ expresses the fact that the purchase of items $A_1, \ldots A_k$ often leads to the purchase of the items $B_1, \ldots, B_l$. Due to this intuitive interpretation, the association rules mining is sometimes called *market basket analysis*. However, the range of the association rule mining is not restricted to this single example. Hipp et al.[HGN00] provide a survey over existing association rule mining techniques.

Outlier mining The *outlier mining* or *anomaly detection* is a further field of Data Mining. The main task of these techniques is the detection of "unusual" objects in a database. Or referring to Hawkings[Haw80], an outlier is defined as: *as an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism*. The outlier detection became a very important technique in the financial industry, where suspicious financial transactions might belong to an illegal action or a fraud attempt. An overview over outlier detection techniques can be found in [NSSH11].

A common feature for all data mining areas is that they are usually meant to be applied to very large databases. The need for efficient algorithms became obvious if we consider that due to Peter Hirshberg (global pulse summit) just in the year 2011 the amount of generated data has exceeded the amount of data generated since the *beginning of the mankind*. And the amounts of gathered and generated data further grow. So, the author of the study THE DIGITAL UNIVERSE IN 2020 predict, that the amount of generated data will double every two year. However, very often the existing Data Mining and Machine Learning techniques are not designed for processing huge amounts of data such that the research in this area is essential. This thesis is meant to

partially fill this gap for some methods of clustering techniques.

1.2 Clustering

Clustering or *unsupervised learning* is a class of approaches for discovering of unknown patterns in the data. The result of a clustering algorithm is a set of object groups, so called clusters, having high similarity among the groups and low similarity between the groups. Different definitions for the inter- and intra-group similarity lead to several clustering models and the decades of research resulted in a variety of approaches. An extensive treatment of all these techniques would require tremendous space and this is not our goal in this thesis. Therefore, in this section we will give only a brief introduction to several classes of clustering approaches before we handle the two special classes - the density-based clustering and projected clustering - in more detail in later chapters of this work.

Due to several aspects the different clustering models are aiming at, a precise categorization of clustering techniques is very difficult. According to Soni et al.[SG12] the clustering techniques can be mainly categorized into *hierarchical* and *partitioning* techniques.

Partitioning methods, as for example the k-means [HK00] or DBSCAN [EKSX96] algorithms, determine a disjunct partitioning of the objects into clusters, i.e., each object is assigned to exactly a single cluster. Soni et al. [SG12] further subdivide the partitioning methods into iterative relocation-based, density-based, and subspace based methods. The main distinctive feature of relocation based approaches is the existence of a parameterized model such that the goal of the clustering is the estimation of optimal parameters with respect to a chosen cost function. For example, the Expectation Maximization (EM) algorithm [DLR77] minimizes the negative log-likelihood of the data points with respect to a mixture of Gaussian distributions, and the probably most well-known k-means algorithm minimizes the squared total distance of all objects to appropriate representative mean objects.

The density-based approaches including DBSCAN or mean shift [FH75] algorithms consider a cluster as a density connected set of objects. The definition of the density term distinguishes the approaches of this class. For example, the DBSCAN algorithm considers a region around a point as dense, if enough neighbors in a specified region are located. The mean shift algorithm utilizes the probabilistic density definition and employs the probability density functions as criterion for merging objects to clusters.

Although assigned to partitioning methods, the subspace based approaches aim at a conceptually orthogonal aspect of clustering, and namely the clustering of objects in high-dimensional spaces. A large number of object's attributes (or dimensions) pose additional challenges for data analysis. Due to the mathematical effect that the distances between any two objects in a high-dimensional space become more and more similar, the determination of meaningful clustering becomes a very challenging task in the full space. Therefore, the goal of subspace clustering approaches is not only to identify the sets of objects but also the relevant attributes for every cluster. The pure subspace clustering methods do not require the object sets to be disjunct, such that an object may belong to multiple clusters. Methods that additionally require a disjunct partitioning of the data belong to a subclass of subspace clustering called *projected clustering*.

In contrast to partitioning techniques, **hierarchical methods** including CURE or BIRCH algorithms provide a cluster hierarchy, that means, the result constitutes multiple clusterings such that the choice of a single particular clustering becomes an additional post-processing step. Depending on whether the clustering is constructed bottom-up or top-down, the hierarchical methods further subdivide into agglomerative and divisive approaches. One of the main advantages of hierarchical methods is the possibility to generate particular clusterings with a low computational overhead. However, this advantage usually arises from a higher clustering complexity.

This thesis deals with effective MapReduce-based solutions for density-based and projected clustering models, and particularly, for the DBSCAN and the P3C algorithms.

1.3 MapReduce

In the past decades, several parallelization models including MPI, OpenMP, Pthreads, CUDA and many more were introduced and intensively used for accelerating computationally complex or data intensive tasks. Moreover, the importance of parallel computing increases more and more, as at the current point of time it seems that physical constraints prevent the further unlimited increase of CPU frequencies. The multitude of existing parallelization models arises from the different types of available parallel computing systems. One possible way to classify parallel systems is the underlying memory location and management. While *shared memory* systems offer a unified memory space for all computing nodes, the *distributed memory* computing requires communication between single nodes for the data exchange. Usually, the shared memory systems are used in integrated high-end server systems that process complex problems with high intra-dependence within the data.
In comparison to distributed memory systems that are usually used in computer clusters consisting of low-price computing nodes, the shared memory systems have a significantly higher price. Moreover, the distributed memory systems allow for a horizontal computational and memory scaling, i.e., the increase of the computational power or memory is achieved through addition of further (inexpensive) computer nodes to the network.

The huge increase of data amounts of the last decades lead to a constant computational power hunger, and the horizontal scalability of distributed memory systems became a huge advantage in many tasks. In the year 2004, Deam and Ghemawat at Google introduced a software framework and programming model called MapReduce [DG04]. The underlying programming model of MapReduce bases on the well-known Bulk Synchronous Parallel (BSP) model and perfectly meets the requirements for fault-tolerant data intensive data processing. Figure 1.2 visualizes the workflow of this framework. As the name suggests, MapReduce consists of two main steps: the map and the reduce phase. In the map phase, the data located in the (usually distributed) file system is read in parallel fashion and an intermediate result is generated. This result, consisting of a set of (key, value) tuples, is

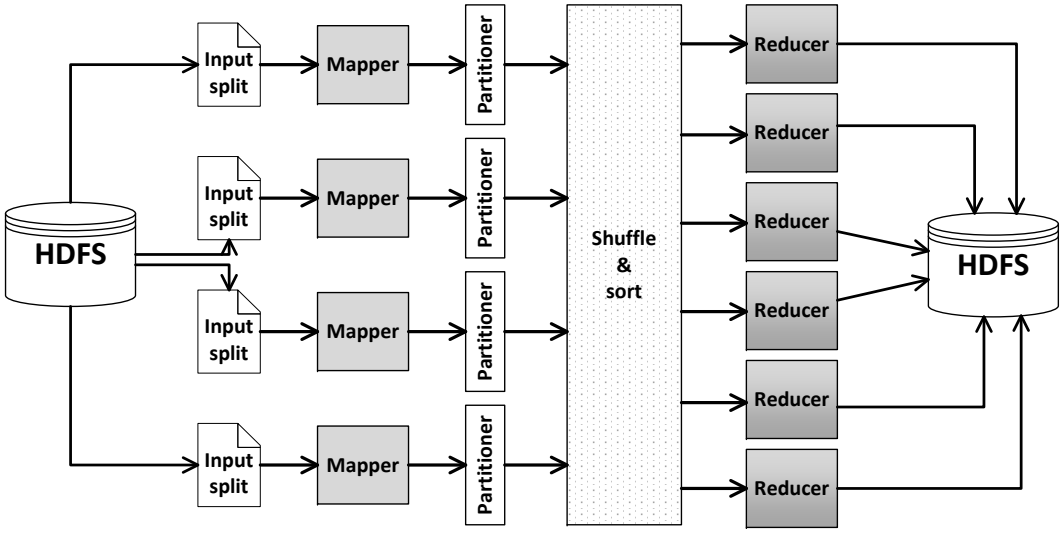


Figure 1.2: MapReduce workflow.

sorted by the key value in the shuffle phase and distributed to the reducers. Each of the value sets with the same key is then independently processed by a single reducer and the final result is written back to the (distributed) file system. Technically, the described workflow further contains facilities for input splitting and mechanism for distributing the data across several reducers (partitioning). The latter is implemented as a function, which computes the reducer ID for each (key, value) pair based on the key value. The default implementation that simply applies the modulo-operator on the hash value of the key, usually provides a sufficiently good workload distribution on the cluster. However, in special situations, as for example in the CC-MR algorithm presented later on, other partitioning techniques are required.

In the MapReduce model, communication between different computing nodes only takes place during a single communication phase, when the intermediate pairs from the map nodes are transferred to the reduce nodes. Apart from this, no further communication takes place. Neither the individual mappers nor the individual reducers communicate with each other. This loose coupling of the computational nodes enables the framework to perform the calculations in a highly distributed and fault-tolerant way. Since all computational nodes process the data independently from each other, the only

limitation for the number of parallel reducer-jobs is the number of unique intermediate key values. Additionally, since the single jobs do not depend on the results of other jobs, the failure of hardware can be easily managed by restarting the same job on another computational node. This high fault-tolerance and the loose coupling of computational nodes suits perfectly for usage of this model on commodity hardware like personal PCs connected to a cluster over a network.

However, this limited communication also poses a challenge for the development of algorithms, which have to be designed such that the data in different mappers/reducers can be processed completely independently. If such a decoupling is not possible, additional information must be communicated to the computing nodes. Often, the same information is required by multiple nodes such that the same pieces of data are replicated multiple times. For huge amount of data, such data replication may become a bottleneck of the overall algorithm.

A general challenge when dealing with parallel programs is an equal workload distribution on the computing nodes. For simple parallelization problems in which the subtasks have a similar workload, MapReduce provides a convenient way to distribute the subtasks. Complex problems, however, often require non-trivial solutions.

Summarized, in the last decade MapReduce proved to be a convenient solution for many data intensive and computational tasks. However, as usual it holds that “there is no free lunch”; the static structure of the MapReduce processing workflow makes it hard to apply in several situations.

1.4 Contributions and structure of the thesis

The remainder of the thesis is divided into two parts.

Part I deals with density-based clustering, to wit with the DBSCAN algorithm in the MapReduce framework. We show how to decompose the original iterative DBSCAN algorithm into multiple phases, such that the result of a sequence of MapReduce jobs results in a correct DBSCAN cluster-

ing. To perform the clustering efficiently, we further propose two grid-based similarity self-join techniques MR-DSJ and PHiDJ for low, medium and high-dimensional spaces in Chapter 3, and a novel algorithm CC-MR for finding connected components in graphs in Chapter 4. We show the superiority of our proposed similarity self-join and connected component approaches, and compare the resulting DBSCAN solution with existing techniques in Chapter 5.

Part II of the thesis addresses the adaptation of the projected clustering approach P3C for large datasets. We will show and explain why the clustering model of the original P3C algorithm is not able to produce a meaningful clustering in the presence of large amount of data and present several extensions and improvements. The resulting P3C$^+$ is then shown to be effective as well as efficient for very large data.

In **Part III**, this thesis is summarized and an outlook is given. Finally, **Part IV** contains the appendices of the work.

1.5 General remarks

To simplify the reading of this work, each chapter contains a table with relevant notions and abbreviations. Moreover, to reduce the repetition of words, we use the terms *point* and *object*, and *vertex* and *node* interchangeably, if there is no risk of confusion.

Part I

Density-based clustering of massive data

As described in the previous chapter, density-based clustering techniques represent an own class of clustering algorithms. The underlying assumption of these models is that a cluster is a dense set of points. Depending on the algorithm, different definitions of density are assumed including for example a minimal number of points in some specified volume as in the DBSCAN algorithm, or probability density of a point set in the mean shift algorithm.

The density-based approaches have several applications, ranging from analysis of spatial relationships in spatial databases to analysis of X-ray crystallography, where the locality of objects plays a central role. One of the main tasks is *class identification*, as for example the identification of land and water areas on satellite images.

In this part, we propose and evaluate a novel parallelized algorithm JBMR-DBSCAN (*Join-Based MapReduce-DBSCAN*) for the DBSCAN clustering for the MapReduce framework. The basic idea of our algorithm is to replace the integrated iterative process of neighbor querying and label assignment for every point in the original DBSCAN algorithm into two phases. The first phase computes all neighbors for every object in the database in parallel, that is, the similarity self-join of the database. The second phase is the determination of connected components in the graph with respect to core/non-core property of the objects that corresponds to labeling of the object by cluster IDs.

The presented solution is a very general approach that can be applied on every kind of data for which a metric distance function is defined, and is one of the key differences to the currently existing approaches that are applicable on vector data only. Due to the component-based structure of the algorithm, its efficiency can be improved by accelerating the single components independently. The efficiency of the presented solution will even more be improved by interconnecting the parts of single components, as we will show later.

This part has the following structure. First, in Chapter 2, we introduce the DBSCAN algorithm and explain the basic idea of our solution in more detail. Chapters 3 and 4 describe efficient similarity self-join techniques for low and medium to high-dimensional vector data and an approach for determining

connected components in graphs, respectively. In Section 5.1, the existing solutions for the DBSCAN approach are described. Our novel JBMR-DBSCAN algorithm and its integrated version are explained in Sections 5.2 and 5.3, respectively. The evaluation of the presented solutions is provided in Section 5.4. Finally, conclusions are summarized in Section 5.5.

Chapter 2

Introduction to DBSCAN and basic idea

Contents

2.1	Sequential DBSCAN algorithm	**19**
2.2	Basic idea of JBMR-DBSCAN	**24**

In this chapter, we first present the DBSCAN [EKSX96] algorithm and its original sequential implementation in Section 2.1. Then, in Section 2.2, the basic idea of our algorithm will be presented.

2.1 Sequential DBSCAN algorithm

Before we start with the actual DBSCAN algorithm, we introduce several notions. Table 2.1 summarizes the used symbols of this chapter.

Definition 2.1 (Metric space) *A metric space (O, d) is an ordered pair, where O is a set (of objects) and $d : O \times O \mapsto \mathbb{R}$ is a metric on O. I.e., for d holds:*

1. *Identity of indiscernibles:* $d(x, y) = 0 \iff x = y$
2. *Symmetry:* $d(x, y) = d(y, x)$

Symbol	Description
(O, d)	metric space
d	metric distance function
DB	database
o, p, q	objects in the database
$NN_\varepsilon(p)$	Neighborhood of point p in the database
$CoreSet_{DB}$	set of core objects
$Border_{DB}$	set of border objects
$Noise_{DB}$	set of noise objects
$DDR(p)$	set of directly density reachable objects of p
$DR(p)$	set of density reachable objects of p
DC_{DB}	set of density connected objects
$\mathbf{C}$	Cluster w.r.t. core-object p
$\mathcal{C}$	DBSCAN clustering
ε	Parameter ε of the DBSCAN algorithm
$minPts$	Parameter $minPoints$ of the DBSCAN algorithm

Table 2.1: Used symbols in this chapter

3. *Non-negativity:* $d(x, y) \geq 0$

4. *Triangle inequality:* $d(x, z) \leq d(x, y) + d(y, z)$

for all $x, y, z \in O$.

From now on, we assume that a finite database $DB \subseteq O$ is given and the two parameters $\varepsilon \in \mathbb{R}$ and $minPts \in \mathbb{N}^+$ of the DBSCAN algorithm are set.

Definition 2.2 (ε neighborhood) *Let $q \in DB$. The ε neighborhood $NN_\varepsilon(q)$ of q is defined as:*

$$NN_\varepsilon(q) := \{p \in DB | d(p, q) \leq \varepsilon\}.$$

Given this definition, we can define the central entity of the DBSCAN algorithm - a *core object*.

Definition 2.3 (Core objects, $CoreSet_{DB}$) *Let the set $CoreSet_{DB} \subseteq DB$ be defined as:*

$$CoreSet_{DB} := \{p \in DB | |NN_\varepsilon(p)| \geq minPts\},$$

where $|\cdot|$ is the cardinality of a set. Then $p \in CoreSet_{DB}$ is called a core object.

A core point is therefore a point that is located in a dense region of the space, i.e., it has sufficiently many (at least $minPts$) objects in its surrounding. Now we proceed with definitions, which are required for describing the cluster and clustering model of the DBSCAN algorithm.

Definition 2.4 (Direct density-reachability, $DDR(p)$) *Let $p \in CoreSet_{DB}$ be a core object and the set $DDR(p) \subseteq DB \times DB$ be defined as:*

$$DDR(p) := \{(p,q) | q \in NN_{\varepsilon}(p)\}.$$

Then the object $q \in DB$ is called directly density-reachable *from p, if $(p,q) \in DDR(p)$.*

A point q is directly reachable from a core point p if it lies in the ε-neighborhood of the object p. Note that q is not necessarily a core object.

Definition 2.5 (Density-reachability, $DR(p)$) *Let $p \in CoreSet_{DB}$. An object $q \in DB$ is called* density reachable *from the core-object p, if $(p,q) \in DR(p)$, where $DR(p)$ is defined as:*

$$\begin{aligned} DR(p) :=& \{(p,q) | q \in DB \wedge \exists n \in \mathbb{N}^{+} \exists p_1 \ldots p_n \in CoreSet_{DB} \\ & \forall i \in [1, n-1] : (p_i, p_{i+1}) \in DDR(p_i) \\ & \wedge (p_n, q) \in DDR(p_n)\} \end{aligned} \tag{2.1}$$

I.e., an object q is density-reachable from p if there is a chain of directly density connected core objects that connects p and q. Again, q is not necessarily a core object.

Definition 2.6 (Density-connection, DC_{DB}) *An object $p \in DB$ is called* density connected *to object $q \in DB$, if $(p,q) \in DC_{DB}$, where DC_{DB} is defined as:*

$$\begin{aligned} DC_{DB} :=& \{(p,q) | p,q \in DB \\ & \wedge \exists o \in CoreSet_{DB} \\ & \wedge (o,p) \in DR(o) \\ & \wedge (o,q) \in DR(o)\} \end{aligned} \tag{2.2}$$

I.e., q and p are density connected if they are density reachable from some core-object o.

Definition 2.7 (Border object, $Border_{DB}$) *The set of border objects of the database DB is defined as*

$$Border_{DB} := \{o \in DB \setminus CoreSet_{DB} | \exists p \in CoreSet_{DB} \wedge o \in DDR(p)\}.$$

Every density reachable object that is not a core object is therefore a border object.

Definition 2.8 (Noise) *The set of noise objects in the dataset DB is defined as:*

$$Noise_{DB} := \{o \in DB \setminus CoreSet_{DB} | \neg \exists p \in CoreSet_{DB} : o \in DDR(p)\}.$$

Finally, every object that is neither a core object nor a border object is a noise object.

Definition 2.9 (Cluster, $\mathbf{C}$) *A cluster $\emptyset \neq \mathbf{C} \subseteq DB$ w.r.t. ε and $minPts$ is a non-empty set of DB that satisfies:*

1. Maximality: $\forall p, q \in DB : (p \in \mathbf{C} \wedge q \in DR(p)) \rightarrow q \in \mathbf{C}$

2. Connectivity: $\forall p, q \in \mathbf{C} : (p, q) \in DC_{DB}$

The previous definitions and the algorithm that we will present later on ensures that the DBSCAN clustering fulfills following properties: the clustering $\mathcal{C}$ of the dataset DB is the smallest set $\mathcal{C} \subseteq 2^{DB}$ with

1. *Cluster disjunction:* $\forall \mathbf{C}_i, \mathbf{C}_j \in \mathcal{C} : \mathbf{C}_i \cap \mathbf{C}_j = \emptyset$

2. *Each objects belongs either to a cluster or is a noise object:* $DB = \mathcal{C} \cup Noise_{DB}$

Listing 2.1: Pseudo-code of the sequential DBSCAN algorithm.

```
1  void dbscan(dataset DB, ε, minPts)
2    mark all objects as unvisited
3
4    while some objects univisited
5      o = random unvisited object
6      o.visited = true
7
8      if NN_ε(o) ≥ minPts // o is a core point
9        create cluster C_new + add o to C_new
10
11       // expand cluster around o
12       N := NN_ε(o) // get neighbors of o
13       foreach p ∈N // N changes over time!
14         if p unvisited
15           p.visited = true
16           if NN_ε(p) ≥ minPts
17             N := N ∪NN_ε(p)
18             if p belongs to no cluster
19               add p to C_new
20
21       output C_new
22     else
23       mark o as noise
```

DBSCAN algorithm The original sequential DBSCAN algorithm presented in [EKSX96] determines the previously defined clustering in an iterative fashion. Its pseudo code is given in Listing 2.1.

The algorithm starts the processing with a random unvisited object o. If o is a core-object all density reachable objects (lines 8–19) of o are collected by iterative neighborhood querying and assignment of the objects to the cluster C_{new}. When the cluster is completely expanded, a new random point o is chosen and expanded in the same way. The computational complexity of the algorithm is hidden in lines 12 and 16 where the neighbor points are determined. Due to the double loop (while-loop and the foreach-loop), that results in a quadratic number of distance computations if no supplementary index structures are employed. For that reason, the most optimization and

parallelization approaches usually focus on this problem and try to accelerate the DBSCAN algorithm by either reducing the number of computations (for example using index structures) or parallelizing this computationally expensive part. Interestingly, the assigning of the objects to the clusters is usually considered as a cheap process and no acceleration of this part is performed. However, assuming a large database and that all distances are already computed, the sequential search of clusters may require a significant amount of work. For that reason, in our upcoming solution, we consider the problem of efficient parallelization of distance computations as well as the parallelized determination of the final clustering result.

2.2 Basic idea of JBMR-DBSCAN

As previously briefly introduced, the basic idea of our parallelized DBSCAN approach JBMR-DBSCAN (*Join-Based MapReduce-DBSCAN*) is to break down the iterative way of clustering into two phases - parallel determination of neighborhood for every database object and the labeling of objects (cluster assignment). The first similarity self-join phase can be calculated by any existing algorithm, like for example [OR11] or one of the approaches that will be presented later on. To accomplish the object labeling task, several solutions exist. The probably simplest solution uses a pre- and a postprocessing step to achieve the goal and is depicted in Figure 2.1. In the preprocessing step, the result of the similarity self-join phase is split up into two edge sets E_{core} and E_{other}. The first one, E_{core}, contains edges between core objects and the second set E_{other} all other edges. In Figure 2.1, E_{core} edges are represented by solid lines and E_{other} by dashed lines. Here, an edge is a pair of objects with a distance smaller than or equal to DBSCAN parameter ε. Intuitively, in the preprocessing step, the sets of core and non-core objects is determined. As described earlier, the non-core objects are either border or noise objects, while core objects constitute the “backbones” of the clusters.
Since a cluster in the DBSCAN algorithm is a set of connected core objects, which is known after the preprocessing step, the problem of finding *minPts*-

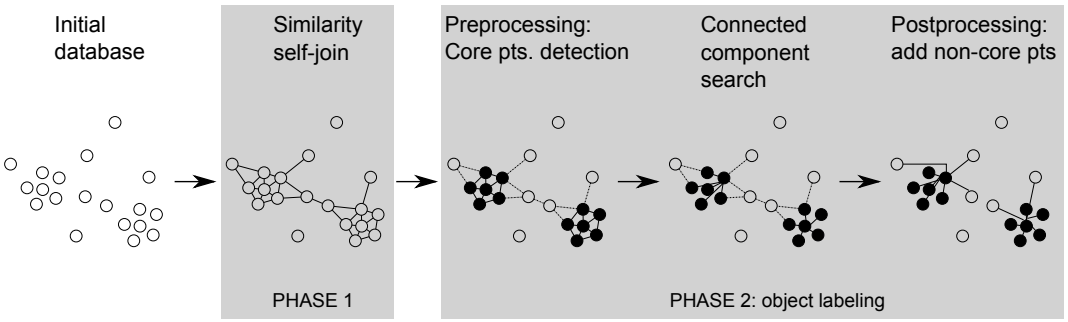


Figure 2.1: Simple DBSCAN solution with pre- and postprocessing steps. Solid edges connect core points (filled circles), while dashed lines represent remaining edges of the graph computed in the similarity self-join step.

constrained components is reduced to the problem of finding components with $minPts = 1$ in the subgraph induced by E_{core} edges (i.e. using only core objects), and assigning border objects in the subsequent step. The determination of connected components with $minPts = 1$ is the standard connected component definition. Therefore, in the second step a standard algorithm for finding connected components can be applied on this graph, the result of which is a set of connected components without non-core points. Finally, in the post-processing step, the non-core objects are assigned to appropriate core objects.The assignment of non-core objects to the clusters is straightforward: if a non-core object nc has no edges, it is marked as noise. If there is exactly one edge to a single core object c, then nc is assigned to the cluster of c. If nc is connected to multiple core objects from different components, every random assignment of nc results in a valid solution.

One disadvantage of the presented solution is that it requires an additional MapReduce job for connecting border objects to the found clusters (the postprocessing step). For that reason, in this thesis, we propose an algorithm for finding connected components with respect to the $minPts$ constraint, i.e., that considers the difference between core and non-core objects and eliminates the need for the postprocessing. The adjusted workflow is shown in Figure 2.2.

Summarized, our novel JBMR-DBSCAN algorithm performs the following

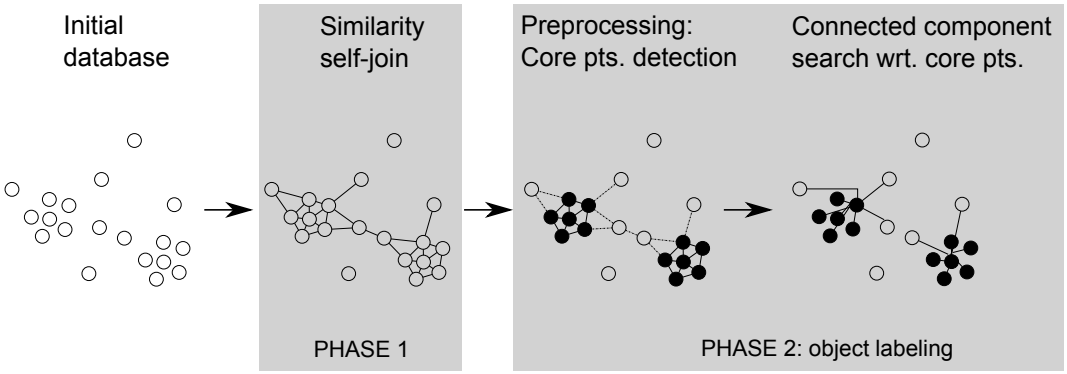


Figure 2.2: Workflow of the DBSCAN solution with connected components algorithm that regards the $minPts$ constraint. The postprocessing step is integrated into connected component search.

steps:

1. Similarity self-join on the database objects
2. Determination of core points
3. Finding connected components with respect to $minPts$ constraint

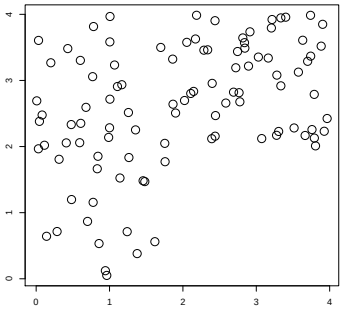
The example in Figure 2.3 once again illustrates the steps of the algorithm. The result of the similarity self-join (Figure 2.3(b)) on the original dataset in Figure 2.3(a) is a graph in which an edge represents a result pair from the join result. We call such a graph a *similarity self-join graph*, and define it in Definition 2.10.

Definition 2.10 (Similarity self-join graph $G_{\bowtie_\varepsilon}$) *Let DB be a database and let $\bowtie_\varepsilon$ denote the similarity self-join operator, i.e., $DB \bowtie_\varepsilon DB = \{(p,q) \in DB \times DB | dist(p,q) \leq \varepsilon\}$. Then the similarity self-join graph is defined as:*

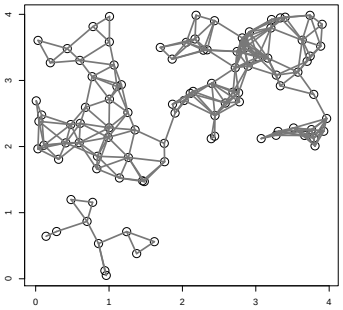
$$G_{\bowtie_\varepsilon} = (DB, DB \bowtie_\varepsilon DB)$$

for a user-defined distance function dist *and parameter $\varepsilon \in \mathbb{R}$.*

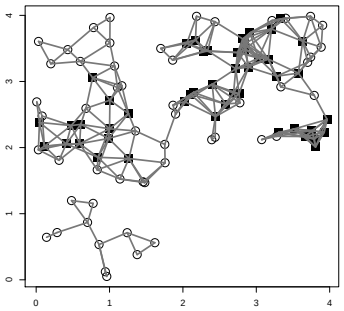
After the second step, the core and non-core objects are identified (Figure 2.3(c)) and finally the connected components with respect to the core and non-core objects are computed (Figure 2.3(d)).



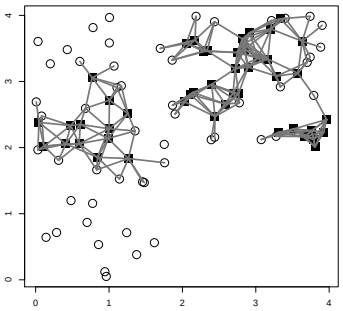
(a) Original dataset.



(b) Similarity self-join graph $G_{\bowtie_\varepsilon}$ for $\varepsilon = 0.5$.



(c) Determined core and non-core objects for $minPts = 7$ (circles = non-core objects, rectangles=core objects).



(d) Final result of connected components algorithm (=clustering result).

Figure 2.3: Phases of our novel JBMR-DBSCAN algorithm.

The presented solution is a feasible and correct implementation of the DBSCAN algorithm that can quite easily be implemented using existing approaches in the literature. In our *integrated JBMR-DBSCAN* algorithm, we furthermore partially combine all steps to save communication and computations in the second and third steps. The details are explained in Section 5.3.

Summary

In this section, we presented the DBSCAN clustering model, the original sequential implementation and explained the general structure of our novel solution. Although the proposed algorithm is applicable for very general metric data, in this work, we focus on vector data, as these are a common representation of the data in a large range of applications. Therefore, in the following chapters, we will first proceed with a novel technique for the similarity self-join step on vector data, and then introduce a novel algorithm for the determination of connected components in graphs. For both approaches we will provide detailed pseudo codes for the MapReduce framework. Finally, in Section 5.2 we will present our MapReduce-based JBMR-DBSCAN algorithm.

Chapter 3

Similarity self-join of vector data

Contents

3.1 Introduction **31**

3.2 Related Work **33**

3.3 MR-DSJ: Efficient Distance Self-Join of low-dimensional data **35**

3.3.1 MR-DSJ algorithm 36

3.3.2 Pruning distance computations 41

3.3.3 Implementation of MR-DSJ algorithm 45

3.3.4 Effectiveness of the MR-DSJ algorithm 47

3.4 Experiments **50**

3.4.1 Scalability on synthetic data 50

3.4.2 Efficiency gain by the $MINDIST$ filters 53

3.4.3 Scalability on real world data 54

3.5 Further Analysis **56**

3.5.1 Data replication and data dimensionality 56

3.5.2 Influence of the parameter ε 57

3.6 PHiDJ: Extending to high-dimensional spaces and skewed data **57**

3.6.1 Reducing communication costs 58

3.6.2 Workload balancing through variable grids . . . 61
3.6.3 Implementation in MapReduce 79
3.6.4 Analysis 84
3.7 Experiments **87**
3.7.1 Synthetic data 89
3.7.2 Real-world data 91
3.7.3 Grid width 91
3.7.4 Variable grid sizes 93
3.7.5 Workload distribution 96
3.8 Conclusion **98**
3.9 Future work **98**

Similarity self-join is a basic database operator that computes all pairs of similar objects of a given database. In this work, the similarity self-join is used for density-based clustering, however, various further data mining algorithms including near-duplicate detection or outlier analysis rely on this fundamental operator.

In this chapter, we investigate two novel approaches for efficient distance-based self-join algorithm for MapReduce on vector data, as vectors are a common representation for various data types. First, in Section 3.3, we present our MR-DSJ algorithm, a single-iteration grid-based approach for low-dimensional spaces. Then, in Section 3.6, we extend the MR-DSJ algorithm by multiple concepts that allow for efficient processing of medium to high-dimensional data, and improve the efficiency for skewed data distributions that results in the PHiDJ algorithm. The experimental evaluation of both approaches in Sections 3.4 and 3.7 for MR-DSJ and PHiDJ, respectively, demonstrate the superiority over other join algorithms for MapReduce.

3.1 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. The vast majority of data analysis approaches, however, sequentially run on single computing node. Processing huge datasets with millions or billions of records will often exceed the computational capabilities of a single computing node 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 the *join* operations. Similarity self-joins, which are a special type of joins, play an important role in the data analysis: data cleaning [CGK06, RRS00], near duplicate detection [XLY08, Mon00], document similarity analysis [BML10] and data mining tasks including density-based clustering like DBSCAN [EKSX96, BBBK00] or OPTICS [ABKS99] and distance-based outlier detection [TXZ06, KNT00] 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 chapter, we study the distributed computation of *distance-based similarity self joins*. In general, a θ-join of two (or more) relations R, S combines tuples $r \in R$, $s \in S$ which fulfill a given join predicate θ on attribute A: $R \bowtie_\theta S = \sigma_\theta (R \times S) = \{(r, s) \mid \theta(r.A, s.A)\}$. Whereas an *equi-join* demands the equality of objects, $r.A = s.A$, the more general concept of a *similarity join* is based on (dis)similarity distance functions $d : U \times U \rightarrow \mathbb{R}_0^+$ for attributes A of domain U. A low distance value $d(r.A, s.A)$ indicates a high similarity of two objects $r.A$, $s.A$, and a distance value of zero means indiscernibility of the objects. A similarity join $R \bowtie_\varepsilon S = \{(r, s) \mid d(r.A, s.A) \leq \varepsilon\}$ returns all pairs of objects (r, s) whose distance of A does not exceed a maxi-

mum dissimilarity threshold, ε, which is called the *range* or the *radius* of the similarity join. In our applications, the domain U is a multidimensional data space $\mathbb{R}^d$, and distance measures include the L_p norms like the Euclidean distance L_2, Manhattan distance L_1 or the Maximum distance L_∞.

In this work, we study the computation of distance-based self-joins for vector data using the MapReduce programming model. First, we propose a grid-based approach MR-DSJ that 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. In the second part, we introduce the PHiDJ algorithm that extends the MR-DSJ for processing of medium- to high-dimensional spaces and includes techniques for dealing with skewed data distributions. Overall, the main contributions of this chapter are the following:

- We propose the MR-DSJ algorithm for efficiently computing the distance-based self-join on vector data using MapReduce.
- We introduce efficient filtering techniques for reducing communication and computation costs.
- We propose the PHiDJ algorithm for efficiently computing the similarity self-join in medium- to high-dimensional spaces using MapReduce.
- We show how our data partitioning approach can be adapted to skewed data by using variable grids.
- We show the effectiveness of the developed approaches by experiments on synthetic and real-world datasets.

The remainder of this chapter is organized as follows: Section 3.2 describes related work. In Sections 3.3, 3.4 and 3.5, we introduce and analyze our join algorithm MR-DSJ, experimentally evaluate it, and provide additional analytical insights in the proposed technique. The PHiDJ algorithm is described in Section 3.6, and evaluated in Section 3.7. Section 3.8 concludes this chapter, and in Section 3.9 the future work is presented.

3.2 Related Work

As described in the introductory section, the distance-based similarity self-join is defined as $R \bowtie_\varepsilon S = \{(r, s) \mid d(r.A, s.A) \leq \varepsilon\}$. The probably simplest solution for its computation is a nested loop over both relations, which, however, has the disadvantage of quadratic complexities for computation and I/O. These problems have led to the development of advanced approaches [Dat06] including block nested loop, sort-merge, hash- or partition-based or index-based techniques which try to alleviate one of these or both problems.

However, not every approach can 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 datasets has often to grapple with the additional problem of not indexed data. Due to the size, and in case of distributed data storage, 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 that often can be performed in an on-line fashion [ZJ03]. Appropriate data partitioning can lead to a significant efficiency gain of a 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 an equi-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]. In the cases when additional information about the data is available, partitioning techniques can also address the data skewness problem [DNSS92].

In general, the parallelization of joins leads to a higher efficiency, and join processing using the MapReduce framework has already found high attention. [BPE+10, DN14] provide an overview of common join strategies in MapReduce. In [PPR+09], MapReduce is compared to 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 address 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. These strategies distribute the smaller join partner, R, to all computing nodes via the distributed file system and send the larger one, S, through the regular input file mechanism to the mappers. For our particular demand of self-joins, however, such a procedure is prohibitive as in this case the entire dataset would have to be replicated to every node.

The field of similarity joins on MapReduce also gained a high attention in the last few years. Some approaches are specialized on a certain data type and exploit the characteristics of this data type: E.g., Zhang et al. developed an approach for spatial joins for two-dimensional complex shape objects [ZHL+09]. Baraglia et al. [BML10] propose a two-step similarity self-join for textual documents based on inverted lists, which uses the *reference tile method* [DS00] for avoiding duplicate results. Afrati et al. [ASM+12] propose a similarity join algorithm, which specializes on string data and data represented as sets. In this approach, duplicate results are avoided using lexicographic orderings. Also the approaches by Vernica et al. [VCL10] and Metwally et al. [MF12] address similarity joins for set and multiset data using MapReduce. Other, more general, join approaches consider vector spaces and metric spaces. Lu et al. [LSCO12] and Zhang et al. [ZLJ12] present k-NN joins for Euclidean spaces using Voronoi based partitioning [LSCO12] and space-filling curves [ZLJ12], respectively.

Okcan and Riedewald [OR11] proposed a very general join approach, which can compute the result of a θ-join for arbitrary join predicates. The authors propose an effective randomized balancing strategy to distribute the computations of all possible result tuples (i.e. each pair of objects) across a given set of reducers. In the reduce phase, an arbitrary join algorithm (which can be chosen by the user) is performed 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. As no assumptions about the type of join are made, every pair

of objects has to be processed by a reducer. Although the authors propose strategies to avoid some computations from the start for special join types, 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 techniques make use of pruning, and the replication of the data is independent from the cluster size.

Silva et al. [SR12] propose a partition-based similarity self-join for metric spaces for MapReduce that is based on the Quick Join algorithm [JS08]. This approach iteratively partitions the data until each partition can be processed on a single reducer and each partitioning step corresponds to one MapReduce job. The join for each single partition is then computed using the Quick Join algorithm. This approach can also be applied to high-dimensional data. Both [SR12] and [ZLJ12] are data partitioning approaches, which require one or multiple runs on the data for gathering several statistics before the join algorithm can start. Additionally, due to the restriction to metric spaces, both approaches cannot exploit specialized vector space properties as it is done in our approaches.

3.3 MR-DSJ: Efficient Distance Self-Join of low-dimensional data

In this section, we present MR-DSJ, our algorithm for a similarity self-join for low-dimensional vector data in MapReduce. We first introduce a basic approach showing the idea of the algorithm in Section 3.3.1. In Section 3.3.2, we propose improved techniques to reduce the number of distance computations and the communication overhead and provide an efficient implementation for the MapReduce framework in Section 3.3.3. The correctness of the approach is addressed in Section 3.3.4. We also give an analytical analysis of the basic approaches, which extends the experimental evaluation from Section 3.4. For easier reading, we summarize the used symbols and notations of this section in Table 3.1.

Name	Meaning
ε	User defined distance threshold
$\|DB\|$	Size of the database
d	Number of dimensions
c_i	Cell with ID i
$NC(c)$	Set of neighboring cells of cell c
$cell(p)$	Cell of point p

Table 3.1: Used notations and names

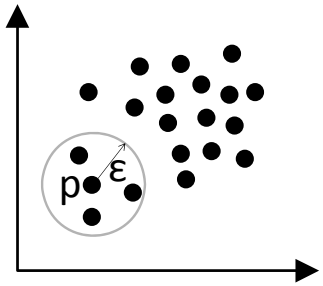


Figure 3.1: Small ε-neighborhood

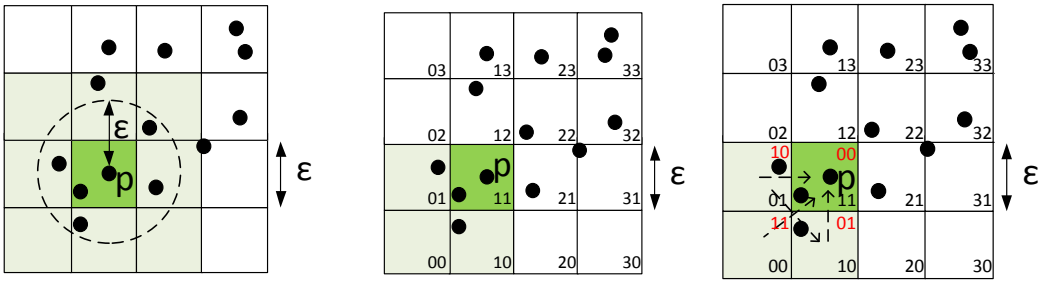
The analytical analysis is performed for the following aspects of the algorithm:

- number of computations per reduce-job [*per-red-comp*]
- input size per reduce-job (communication) [*per-red-input*]
- memory footprint per reduce-job [*mem-footprint*]
- overall number of computations [*overall-comp*]
- overall communication of the algorithm [*overall-input*]

The abbreviations in the square brackets are used in the following for referencing the single analysis aspects. We additionally analyze the worst-case scenario and that way we reason about possible bottlenecks and the load balancing properties of the approach.

3.3.1 MR-DSJ algorithm

In a non-parallelized environment, the naive way for calculating a join is a nested loop that computes the distances between all pairs of points. In the



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

Figure 3.2: Basic ideas of the MR-DSJ algorithm.

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 Figure 3.1 has only a small subset of database objects in its ε-neighborhood but the nested loop solution would calculate the distances to all points. In order to reduce the number of computations, we use a quite common grid-based partitioning approach with equi-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 Figure 3.2(a) for the 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.

The presented approach can be easily translated into a MapReduce program. Each reducer R_i is responsible for one cell c_i and its neighboring cells $NC(c_i)$, and computes via a 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 point in the dataset the distances to all objects in its neighbor cells are computed in some reducer, this simple method is a correct similarity self-join implementation. In the case, when ε is small in comparison 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 smaller or equal ID in every dimension, as shown in Figure 3.2(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 many result pairs are duplicated, which occurs when two objects p and q from neighboring cells are processed in two separate reducers. This is for example 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

	00	01	10	11
00	X	X	X	X
01		-	X	-
10			-	-
11				-

Figure 3.3: Bitcodes for the 2d case

has to differentiate between the points from the different cells that were sent to it. Therefore, we introduce a "bitcode" 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 bitcodes 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 bitcode '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 Figure 3.2(c), where the bitcodes for the cells that are sent to the reducer R_{11} are presented. For example, the lower left cell is assigned the bitcode '11' because it differs from the home cell in both dimensions. Using these bitcodes 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 Figure 3.3, we exemplary show the decision matrices for the 2-dimensional case. For each pair of cells (represented by their bitcodes), a '$\times$' 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 bitcodes 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 Figure 3.2(c), we just have to compute the distances between the cells with the bitcodes '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 3.3.4.

Using this approach, our join algorithm is guaranteed to produce no duplicate result pairs. This does not only save unnecessary distance computations, but also the need to eliminate duplicates after the join.

Analysis In the following we assume that the data is uniformly distributed in a d-dimensional space and the domain of each attribute is [0; 1], i.e., the number of the set of all cells in the grid is equal to ε^{-d} 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) *per-red-input* and the memory footprint of a single reducer *mem-footprint* 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 bitcode information the memory consumption can be halved to $2^{d-1} \cdot C$. A precise description of this reduction technique is presented in Section 3.3.3. The overall communication *overall-input* of the job is equal to $\varepsilon^{-d} \cdot$ *per-red-input* $= 2^d \cdot |DB|$.

Each reducer performs $\sum_{i=0}^{d} C \cdot \binom{d}{i} \cdot \sum_{j=0}^{d-i} C \cdot \binom{d-i}{j} = C^2 \cdot \sum_{i=0}^{d} \binom{d}{i} \cdot 2^{d-i} \cdot 1^i = 3^d \cdot C^2$ computations, such that the overall computations *overall-comp* is equal to $3^d \cdot \varepsilon^d \cdot |DB|^2$. The first sum and the binomial term enumerate the cells C_x with i ones in the bitcodes, while the second sum and the binomial represent

the cells with $d - i$ ones, and namely exactly the cells those bitcodes do not have 1s in the same positions as the cells from C_x. This means, this formula computes the number of computations performed by the MR-DSJ with consideration of the bitcode pruning.

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 *overall-input* is $2^d \cdot |DB|$. Additionally, each of these reducers has to store the complete database locally, such that *per-red-input* and *mem-footprint* grow to $|DB|$. The overall number of computations *overall-comp* is then equal to $\frac{|DB|^2}{2}$. To deal with these extreme case, one possible solution is to combine the MR-DSJ algorithm with the θ-join algorithm in such a way that the join of objects in cells with a very large number of elements is processed in parallel and not in a single reducer. However, in this work, we do not treat this idea in detail.

3.3.2 Pruning distance computations

The basic solution provides a very efficient solution for grid-based similarity self-join on MapReduce as every distance computation is performed exactly once due to the employed bitcode filter. In this section, we introduce two techniques to save even more distance computations and reduce replication by considering the positions of points *within* a cell as will be shown in Sections 3.3.2.1 and 3.3.2.2.

3.3.2.1 Reducer side pruning by MindistCell

We start with an example in Figure 3.4. The cell with bitcode '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*

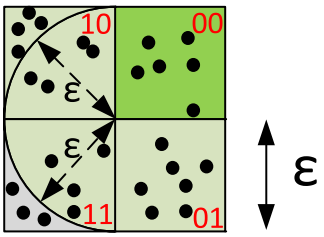


Figure 3.4: Pruning distance computations by $MindistCell$ (for L_2 norm)

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 these 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 3.1 ($MindistCell$) *The distance of a point q to a cell c is defined as*

$$MindistCell(q,c) = \sqrt[p]{\sum_{i=1}^{d} \begin{cases} |lb_c[i] - q[i]|^p & q[i] < lb_c[i] \\ 0 & lb_c[i] \leq q[i] \leq ub_c[i], \\ |q[i] - ub_c[i]|^p & q[i] > ub_c[i] \end{cases}} \quad (3.1)$$

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 bitcodes c_1 and c_2 differ in at least two dimensions (for one dimension, $MindistCell$ is always $\leq \varepsilon$). 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.

3.3.2.2 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 3.1 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 1.

Lemma 1 (Contribution of dimensions) *For a point q from cell c_q, and a cell c, only dimensions which are different in the bitcodes for c_q and c contribute to $MindistCell(q, c)$.*

Proof 3.1 *According to the definition of the bitcode, if the values of the bitcode are equal in a dimension i, then the range of the cells in i is equal. Therefore, the middle rule of definition 3.1 applies and such a dimension does not contribute to $MindistCell(q, c)$.* ■

Theorem 3.1 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 $MindistCell(q, c_j) \leq \varepsilon$.*

Proof 3.2 *Let $b^q = b_1^q, \cdots, b_d^q$ be the bitcode of $cell(q)$ and $b^{c_j} = b_1^{c_j}, \cdots, b_d^{c_j}$ the bitcode of cell c_j. Additionally let I be the set $\{i | 1 \leq i \leq d\}$ with $b_i^q = 1$. Then $|I|$ corresponds to the number of positions in which b^q differs from the bitcode 0^d of the home cell and represents the dimensions which contribute to the $MindistCell$ according to Lemma 1. Let now consider the cell c_j: due to the*

1s-rule (cf. Section 3.3.1) 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 $MindistCell(q, c_j) > MindistCell(q, c_{home})$. ■

3.3.2.3 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 3.2.

Definition 3.2 (MindistPair) *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 bitcode pruning. Further, let c_{home} be the home cell, $md_o = MindistCell(o, c_{home})$ and $md_q = MindistCell(q, c_{home})$. Then $MindistPair(q, o)$ is defined as:*

$$MindistPair(q, o) := \sqrt[p]{md_q^p + md_o^p}, \tag{3.2}$$

for $p \in \mathbb{N}$.

Definition 3.2 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 3.2 (Correctness of MindistPair pruning) *Let q, o, c_q, c_o, c_{home}, md_q, md_o be defined as in Definition 3.2. 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 Figure 3.5 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

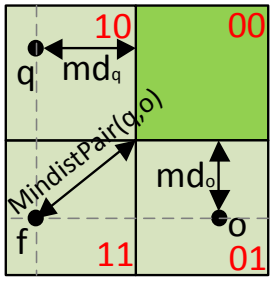


Figure 3.5: Example for MindistPair.

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 3.3 (MindistPair) *According to Definition 3.2, 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 Figure 3.5 i is the vertical axis (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 1, the dimensions which do not differ do not contribute to the $MindistCell$ and, therefore, the proof also holds for this case.* ■

3.3.3 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 3.1 and 3.2 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 the pseudo-code. The recursion in the mapper (Listing 3.1) 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 bitcode. 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 bitcode 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 3.2) cascades **for** and **if** statements to realize the respective loops and pruning strategies introduced in Section 3.3.2. 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 bitcode 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 bitcodes 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 bitcode 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 bitcodes is precomputed in advance

Listing 3.1: The MR-DSJ mapper recursively assigns objects $(id, coord)$ to neighboring partitions pid. The bitcodes 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.

```
1  void DSJ_map(int id, float[] coord)
2    for dim = 1..dimension do
3      home_slice[dim] = int(coord[dim] / ε);
4      dist[dim] = (home_slice[dim]+1) * ε − coord[dim];
5    map_recursive(1, 0, 0, 0.0, id, coord);
6
7  void map_recursive(int dim, long pid, int c, float mdp, int id, float[] coord)
8    if (dim ≤ dimension)
9      pid = (pid << bits_per_dimension) + home_slice[dim];
10     map_recursive(dim+1, pid, c<<1, mdp, id, coord);
11
12     mdp = mdp + dist[dim]^p;
13     if (mdp ≤ ε^p)
14       map_recursive(dim+1, pid+1, (c<<1)|1, mdp, id, coord);
15   else
16     emit(pid, (c, mdp, id, data));
```

by $maxcode = 2^{dimension-1}$, which may be implemented by using the bitshift operator $<<$, i.e., $maxcode = 1 << (dimension - 1)$.

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. 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 bitcode in order to sort the input with respect to ($partition_id$, $bitcode$) in lexicographic order.

3.3.4 Effectiveness of the MR-DSJ algorithm

In this section, we show the correctness, completeness and minimality of the MR-DSJ algorithm. For that, we prove the following lemmata that on their side prepare Theorem 3.3, which states the desired properties.

Listing 3.2: The MR-DSJ reducer computes the join results. The bitcodes 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.

```
1  void DSJ_reduce(long partition_id, Iterator values)
2    cellBuffer.clear();
3    foreach (c_n, mdp_n, id_n, data_n) in values
4      for (c_b=0; c_b < maxcode; c_b++)
5        if (c_n & c_b == 0)   // bitcode filter
6          if (mindist(data_n, c_b) ≤ ε) // mindistCell filter
7            foreach (mdp_b, id_b, data_b) in cellBuffer[c_b]
8              if (mdp_n + mdp_b ≤ ε^p) // mindistPair filter
9                if (d(data_n, data_b) ≤ ε) // join condition
10                 emit(id_n, id_b);
11                 emit(id_b, id_n); // symmetric pair, if desired
12     if (c_n < maxcode)    // relies on secondary sort
13       cellBuffer[c_n].insert((mdp_n, id_n, data_n));
14     if (c_n == 0)
15       emit(id_n, id_n);    // reflexive pair, if desired
```

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. 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 (slice = a row in the grid) in dimension i to which DSJ_map assigns $data_p$ and $data_q$, respectively. Furthermore, let c_p^i, c_q^i denote the bitcodes of p, q in a slice of dimension i where p, q are present.

Lemma 2 (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 3.4 *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.* ■

Lemma 3 (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 3.5 *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 bitcodes c_p^i, c_q^i:*

(i) $(c_p^i, c_q^i) = (1, 1)$*: 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 $c_p^i = c_q^i = 0$ and case (iv) applies.*

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

(iv) $(c_p^i, c_q^i) = (0, 0)$*: 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 $\hat{s}^i < s_p^i$ nor in subsequent slices $\hat{s}^i > s_p^i + 1$, 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. ■

Lemma 4 (Correctness of DSJ_reduce) *MR-DSJ does not emit false positive pairs: $out_{DSJ} \subseteq R \bowtie_\varepsilon R$.*

Proof 3.6 *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 the proposition* $out_{DSJ} \subseteq R \bowtie_\varepsilon R$ *holds.* ■

Theorem 3.3 (Effectiveness of MR-DSJ algorithm) *The algorithm MR-DSJ produces complete and correct results without duplicates:* $out_{DSJ} = R \bowtie_\varepsilon R$.

Proof 3.7 *The completeness of MR-DSJ,* $out_{DSJ} \supseteq R \bowtie_\varepsilon R$*, follows from Lemmata 2 and 3, and the correctness of MR-DSJ,* $out_{DSJ} \subseteq R \bowtie_\varepsilon R$*, holds due to Lemma 4. The absence 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 3.* ■

3.4 Experiments

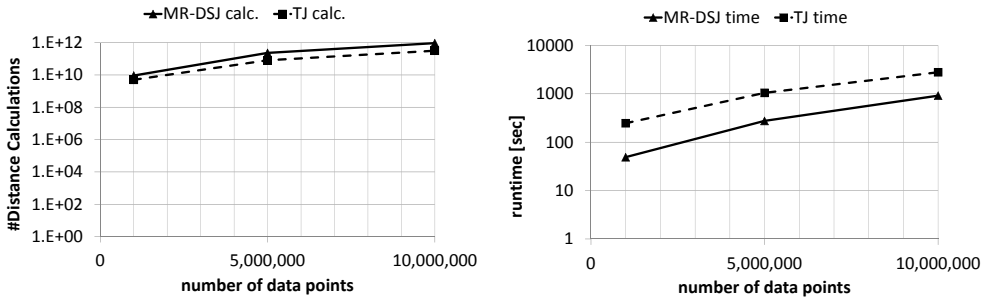
In this section, we present an experimental evaluation of our MR-DSJ approach. In Section 3.4.1, we evaluate its scalability on synthetic data and compare it to the θ-join approach from [OR11]. In [OR11] the author propose the usage of specialized join algorithms inside the reducer tasks. Therefore, we implemented RSJ join [BKS93], which is very well suited for low- to medium-dimensional vector data.

In Section 3.4.2, we evaluate the efficiency gain of the optimizations presented in Section 3.3.2 compared to the basic MR-DSJ algorithm. In Section 3.4.3, we evaluate our approach on real-world datasets.

All the experiments were conducted 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, as well the number of the performed distance computations as the runtime is measured. Runs of the algorithms were aborted if they did not finish within 8 hours.

3.4.1 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



(a) Database size vs. number of distance calculations

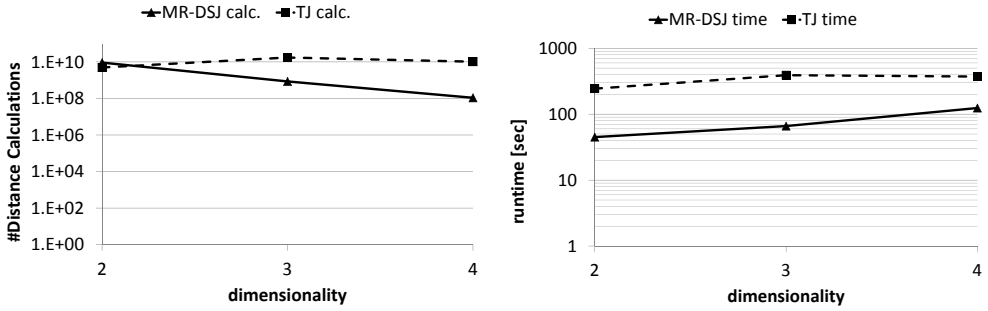
(b) Database size vs. Runtime

Figure 3.6: Influence of database sizes.

values are equally distributed between 0 and 1.

In our first experiment, the database sizes of our synthetic datasets vary 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 Figure 3.6(a) (number of calculations) and Figure 3.6(b) (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.

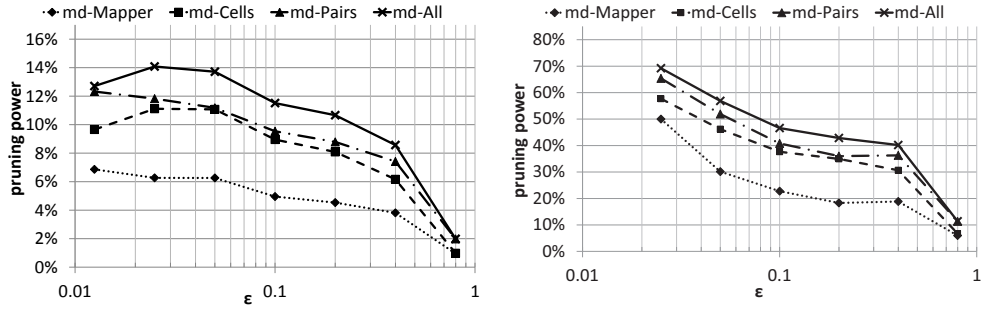
In the next experiment, we vary the dimensionality of our datasets from 2 to 4 dimensions. All datasets consist of approximately 1 million points and are processed with the parameter $\varepsilon = 0.05$. The results presented in Fig-



(a) Dimensionality vs. number of calculations

(b) Dimensionality vs. Runtime

Figure 3.7: Influence of dataset dimensionality.

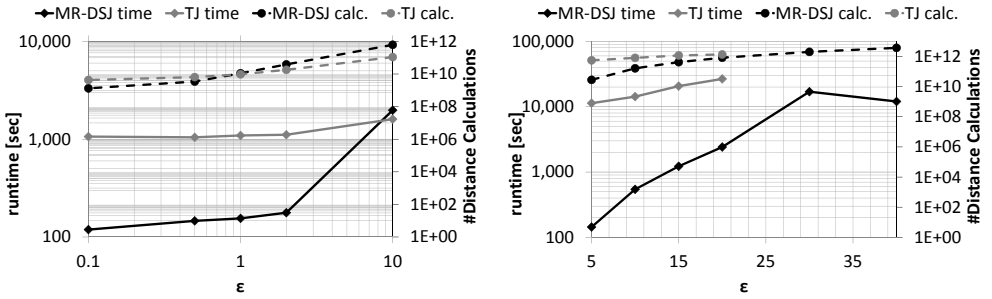


(a) Efficiency gain by the optimizations on a 2d dataset

(b) Efficiency gain by the optimizations on a 4d dataset

Figure 3.8: Efficiency gains on different datasets.

ure 3.7(b) show that the runtimes for both algorithms increase with higher dimensionality, while the runtimes of the MR-DSJ approach are always significantly lower than those of TJ. For the θ-join, the number of distance computations shown in Figure 3.7(a) 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



(a) Varying ε on the cloud dataset (b) Varying ε on the minutiae dataset

Figure 3.9: Influence of ε parameter.

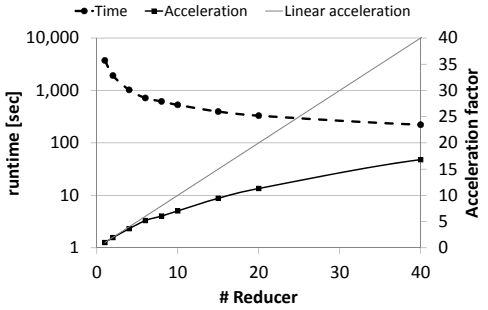


Figure 3.10: Scalability on the minutiae dataset

reducers, which is caused by the higher replication factor.

3.4.2 Efficiency gain by the $MINDIST$ filters

In this section, we evaluate the efficiency gain by the optimizations from Section 3.3.2. 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 Figure 3.8(a) (for the 2-dimensional dataset) and 3.8(b) (for the 4-dimensional dataset).

First, we evaluate the efficiency gain by using the mapper side pruning by computing the MindistCell (cf. Section 3.3.2.2) (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. 3.3.2.1) (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 3.3.2.3 (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 Figure 3.7(a).) 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.

3.4.3 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*. 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 finger-

*`http://cdiac.ornl.gov/ftp/ndp026c/`

print datasets "NIST Special Database 14" and "NIST Special Database 29[†]". It contains ca. 11 million three-dimensional entries, distributed in the ranges [0;832], [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 Figure 3.9(a) and Figure 3.9(b), 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 (Figure 3.9(a)) 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 (Figure 3.9(b)), our observations are similar to 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 (Figure 3.10), we analyze the scalability of our approach on the minutiae with respect to a varying cluster size. Therefore we vary the number of used reducers from 1 to 40. In Figure 3.10, 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

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

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. In our experiments, the usage of 40 reducers results in a 17 times speedup of runtime.

3.5 Further Analysis

In the previous sections, we analyzed the basic MR-DSJ algorithm in terms of number of calculations, communication and memory footprint. 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 these problems in future work.

3.5.1 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, the MR-DSJ approach produces 2^d replica 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 as principal component analysis (e.g. from the Mahout[‡] project) to obtain a reasonably low dimensionality. Mahout is a library for machine learning algorithms for several parallel frameworks including MapReduce.

[‡] `https://mahout.apache.org/`

3.5.2 Influence of the parameter ε

In the analysis part of Section 3.3, we already mentioned the worst-case scenario for the MR-DSJ approach: all points are located in a single grid cell and thus are processed by a single reducer. This case occurs for strongly 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 such a 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.

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 that corresponds to a small number of created reducer 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, is adjusting the cell width to larger or smaller values than ε as happens in our PHiDJ-approach that is presented in the following Section 3.6.

3.6 PHiDJ: Extending to high-dimensional spaces and skewed data

As described earlier, one of the most severe problems of grid based join techniques in MapReduce is the high data replication. In the naive case, when for each home cell all neighbor cells are considered, the data is replicated 3^d times, since information from each cell has to be communicated to all neighboring cells. Exploiting symmetries alleviates the problem, but a 2^d replication of the data as in our MR-DSJ approach remains. High replication leads to high execution times and is often the main bottleneck of an algorithm. Due to the slow increase of the network communication speed

R	database relation
D	set of dimensions = "dimension group"
k	number of dimension groups
$\mathcal{D}$	set of dimension groups
ε_k	ε query radius in a single dimension group
$dist$	an L_m norm based distance function

Figure 3.11: Used symbols in this work.

in comparison to CPU power, the development of low-communication join techniques is indispensable for efficient join algorithms.

The basis of our novel approach *PHiDJ* is the MR-DSJ algorithm described in the previous sections. We build up on the proposed filtering and duplicate prevention techniques, which resulted in efficient join processing for low dimensional spaces and extend it for higher dimensionalities in Section 3.6.1 by reducing its high communication overhead.

In Section 3.6.2, we tackle the problem of skewed data distributions by proposing an efficient technique for dealing with variable grid sizes. The implementation of our techniques is provided in Chapter 3.6.3. In Table 3.11, we summarize the most important symbols used in this section (in addition to the symbols from Table 3.1).

3.6.1 Reducing communication costs

To avoid the problematic high replication of MR-DSJ, we exploit a simple yet powerful idea - instead of performing the join $R \bowtie_\varepsilon R$ in the full d-dimensional space, we split the dimensions in k *dimension groups* (Definition 3.3) containing only disjoint subsets of dimensions and perform the join in these dimension groups. In the following, we use the following notation for the symbols: scalars are denoted by lower case characters (e.g. k), vectors by bold lower case characters (e.g. $\mathbf{p}$), sets of objects by capital letters (e.g. D) and set of sets by script capital letters (e.g. $\mathcal{D}$).

Definition 3.3 *(Dimension groups) Let $\mathbb{R}^d$ be the domain of the given database and $\mathcal{D} = \{D_1, \ldots, D_k\}$ be a set of disjoint subsets of attributes with $\dot{\bigcup}_{i=1}^{k} D_i =$*

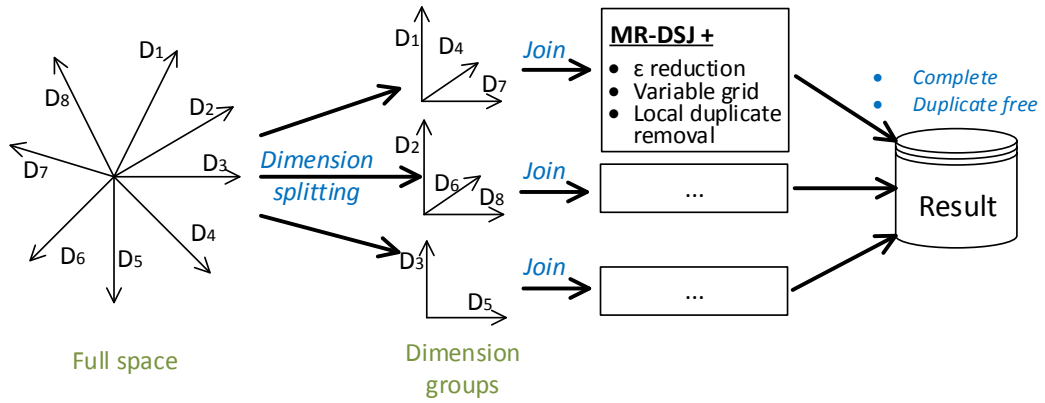


Figure 3.12: Basic idea of our algorithm. All steps are implemented as single MR-jobs.

$\{1, \ldots, d\}$. *We call* D_i *a* dimension group.

A sketch of our complete algorithm is depicted in Figure 3.12. An exemplary d-dimensional space is projected to multiple different-sized subspaces. In each of the subspaces, the MR-DSJ algorithm is performed and the candidate sets of objects are calculated. During the integrated refinement process, potential duplicate pairs are removed. The union of the partial results is then the correct similarity self-join result. All these steps are implemented as a single MapReduce job that is presented in Section 3.6.3. As we will show in Section 3.6.4, splitting the d dimensional space into k subsets leads to a much smaller data replication $k \cdot 2^{\frac{d}{k}}$ instead of 2^d at cost of additional computations. Due to the higher CPU speed in comparison to network communication speed, the overall runtime decreases such that the processing of much higher-dimensional spaces becomes feasible.

Theorem 3.4 states that joining in subspaces leaves the result of the join complete and only adds additional candidate pairs. Our algorithm performs an additional check and eliminates duplicate candidate pairs, and thus is a correct and complete join implementation. In the following, we always assume that $dist_{S \subseteq \{1 \ldots d\}}(\cdot, \cdot)$ is an L_m-norm based distance for $m \geq 1$ in the subspace S, i.e., $dist_S(\mathbf{p}, \mathbf{q}) = \sqrt[m]{\sum_{i \in S} |p_i - q_i|^m}$. $dist_{\{1 \ldots d\}}(\cdot, \cdot)$ is the distance in the full dimensional space and $dist_{D_i}(\cdot, \cdot)$ is the distance restricted to dimensions in the dimension group D_i.

Theorem 3.4 *Let* $R \bowtie_\varepsilon R = \{(\mathbf{p},\mathbf{q}) \,|dist_{\{1\ldots d\}} \leq \varepsilon\}$ *and* $R \bowtie_{D_i} R = \{(\mathbf{p},\mathbf{q}) \mid dist_{D_i}(\mathbf{p},\mathbf{q}) \leq \varepsilon\}$. *Then it holds:* $R \bowtie_\varepsilon R \subseteq \bigcup_{D_i \in \mathcal{D}} \{R \bowtie_{D_i} R\}$.

Proof 3.8 *In order to prove the assumption, one has to show:*

$$(\mathbf{p},\mathbf{q}) \in R \bowtie_\varepsilon R \Rightarrow (\mathbf{p},\mathbf{q}) \in \bigcup_{D_i \in \mathcal{D}} \{R \bowtie_{D_i} R\}.$$

By definition of $dist_S(\cdot,\cdot)$, *every term* $|p_i - q_i|^m$ *only contributes positive (or zero) value to the overall result. I.e., if* S *is a set of dimensions and* $T \subset S$ *then it holds:* $\sum_{i\in T} |p_i - q_i|^m \leq \sum_{i \in S} |p_i - q_i|^m$. *Taking into account that the square root is a monotonically increasing function, it holds:* $dist_T(\mathbf{p},\mathbf{q}) \leq dist_S(\mathbf{p},\mathbf{q})$.

The presented idea, however, still suffers from two drawbacks: It leads to an unnecessarily large number of candidate pairs and it produces duplicate result pairs. Both problems and respective solutions are discussed in the following sections.

Reducing candidate pairs As Theorem 3.4 states, the union of the join results from the subspaces provides a superset of the join result in the full space. Theorem 3.5 states that shrinking the threshold ε to $\varepsilon_k = \frac{\varepsilon}{\sqrt[m]{k}}$ for every dimension group retains the correctness of the join and as a consequence reduces the number of candidate pairs in each dimension group.

Theorem 3.5 *Let* $D_1 \ldots D_k \in \mathcal{D}$ *be defined as in Definition 3.3. Then it holds:*

$$dist_{\{1\ldots d\}}(\mathbf{p},\mathbf{q}) \leq \varepsilon \Rightarrow \exists D_i \in \mathcal{D} : dist_{D_i}(\mathbf{p},\mathbf{q}) \leq \varepsilon_k,$$

where $\varepsilon_k = \frac{\varepsilon}{\sqrt[m]{k}}$.

Intuitively, it means that if $(\mathbf{p},\mathbf{q})$ is a result pair in the full space $\mathbb{R}^d$ then there is at least one subspace D_i in which $(\mathbf{p},\mathbf{q})$ will be a result pair w.r.t. ε_k.

Proof 3.9 *Let* $D_1, \cdots, D_k$ *be the* k *chosen dimension groups of the full space* $\mathbb{R}^d$. *Then it holds:* $R \bowtie_\varepsilon R \subseteq \{(\mathbf{p},\mathbf{q}) \,|\, \sum_{D_i \in \mathcal{D}} \sum_{j \in D_i} |p_j - q_j|^m \leq \varepsilon^m\}$. *Let*

define $a_i = \sum_{j \in D_i} |p_j - q_j|^m$.
Proof by contradiction: assume that all distances a_i *in the subspaces* D_i *are greater than* ε_k^m. *Then it holds* $k \cdot a_i > k \cdot \varepsilon_k^m = k \cdot \frac{\varepsilon^m}{\sqrt[m]{k}^m} = \varepsilon^m$. *Contradiction to the assumption that* $dist_{\{1...d\}}(\mathbf{p}, \mathbf{q}) \leq \varepsilon$. *I.e., there must be at least a single* D_i *with* $a_i \leq \varepsilon_k^m$. ■

Duplicates handling Duplicate result pairs $(\mathbf{p}, \mathbf{q}) \in \bigcup_{D_i \in \mathcal{D}} R \ltimes_{D_i} R$ occur if $(\mathbf{p}, \mathbf{q})$ is considered as a candidate in at least two dimension groups D_i and D_j. In this case, in each dimension group the pair $(\mathbf{p}, \mathbf{q})$ is emitted as a result. To avoid this, we introduce an ordering of dimension groups and check (i.e. compute the distance) for each candidate pair $(\mathbf{p}, \mathbf{q})$ in dimension group D_i whether there is a smaller dimension group $D_{smaller} < D_i$ in which $(\mathbf{p}, \mathbf{q})$ is also emitted. If this is the case, the join algorithm in D_i does not emit $(\mathbf{p}, \mathbf{q})$. This procedure avoids duplicates at the cost of additional distance calculations. A precise cost analysis is given in Section 3.6.4.

3.6.2 Workload balancing through variable grids

The efficiency of grid-based join techniques like MR-DSJ is highly dependent on the equal data distribution onto grid cells. Skewed data distributions lead to unbalanced cell load and thus to a high communication and computational workload of single reducers. Figure 3.13 depicts such a workload disbalance for the MR-DSJ algorithm with a static grid on a real world dataset, whose marginal distributions are shown in Figure 3.14. A point in the plot (Figure 3.13) corresponds to a single reducer, the value of the x-axis is the number of input points, and the value on the y-axis is the number of distance computations performed by a reducer in the cell. As one recognizes from marginal distributions in Figure 3.14, the used dataset is (closely) normally distributed in every dimension that results in a very uneven distribution of reducer inputs (=number of objects per cell) and the number of performed distance computations in every cell. While most of the reducers perform only few computation, there are two cells (points in the top right corner in Figure 3.13) in which significantly more computations have to be performed. I.e.,

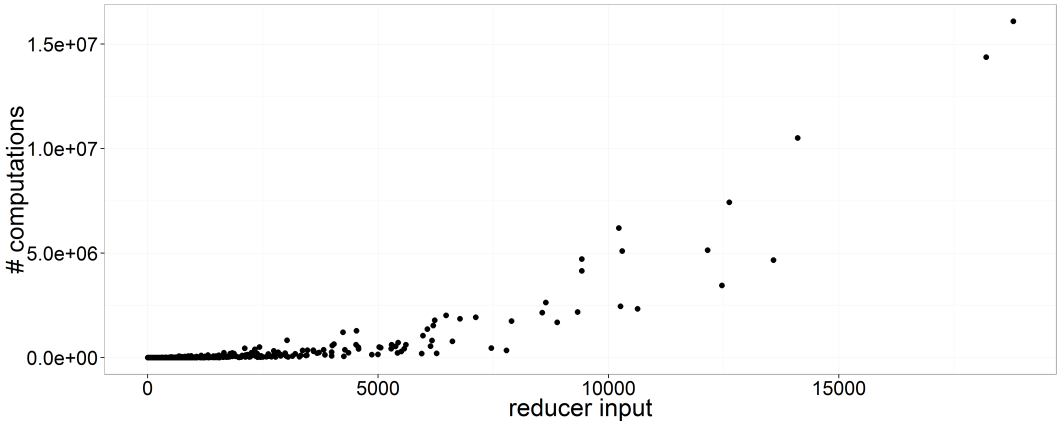


Figure 3.13: Reducer input vs. number of distance computations on skewed dataset (static grid, Flickr dataset)

the runtime of the overall parallel algorithm will be dominated by those two outliers (assuming a sufficiently large number of parallel tasks).

For that reason, in this section, we consider several techniques to adjust the cell sizes of the employed grid in order to improve the workload balancing properties of the algorithm (Section 3.6.2.2). We define a cost function for the join processing in Section 3.6.2.1, which is then used to evaluate the employed heuristic approaches on several databases (Section 3.6.2.4). Section 3.6.2.3 deals with the integration of variable grids into the PHiDJ algorithm (or to be more precise into the MR-DSJ parts of the PHiDJ algorithm).

3.6.2.1 Join processing costs

Intuitively, the costs (e.g. running time) for the join processing mainly comprise the communication, number of computations, and costs for writing the results to the distributed file system. Further, depending on the regarded kind of costs (for example for running time), a distinction between the overall distance computations and communication, and maximal distance computations and communication of a single reducer is meaningful as both factors contribute to the overall costs of parallel algorithms. The optimization of the single aspects, i.e., the overall communication or maximal distance

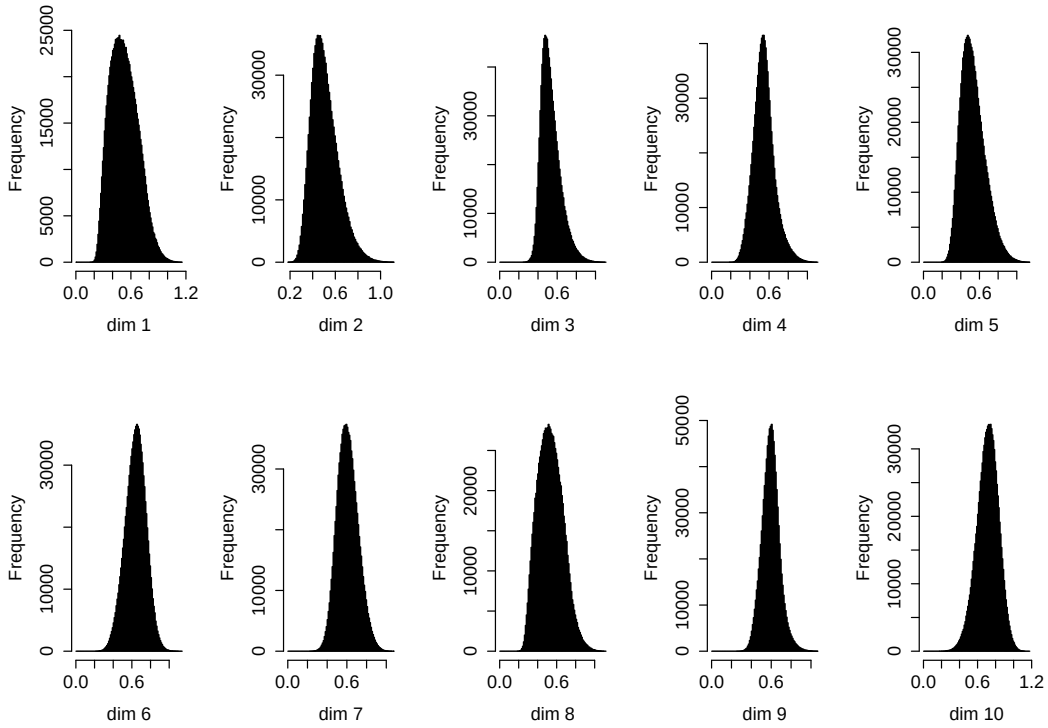


Figure 3.14: Marginal distributions of the dataset used in Figure 3.13.

computations is simple. For example, for reducing the overall communication a grid with a single cell is optimal since in this case no communication is needed. And for reducing the maximal number of distance computations, a grid with as few elements as possible is desirable, since no object pairs must be filtered. However, for optimization of the overall costs, a combined criterion is needed. Let $\mathcal{P}$ be an arbitrary grid partitioning, then we define the overall costs of the join as:

$$cost(\mathcal{P}) := \frac{1}{Z} \cdot (\frac{c_{dist} \cdot dist_{all}(\mathcal{P}) + c_{out} \cdot output_{all}}{\#reducer} + c_{comm} \cdot comm_{all}(\mathcal{P}) + c_{dist} \cdot dist_{max}(\mathcal{P}) + c_{comm} \cdot comm_{max}(\mathcal{P}) + c_{out} \cdot output_{max}) \tag{3.3}$$

where c_{dist}, c_{comm}, c_{out} are the respective costs for a single distance computation, for communicating a single object, and the cost for writing a single ob-

ject to the distributed file system. $dist_{all}(\mathcal{P})$, $comm_{all}(\mathcal{P})$ and $output_{all}$ denote the overall number distance computations, overall input size of all reducers and overall output of all reducers, respectively. Consequently, $dist_{max}(\mathcal{P})$, $comm_{max}(\mathcal{P})$ and $output_{max}$ stand for maximal number of distance computations, etc., of a single reducer. The variable Z is a normalization constant for better comparability of the results that describes the costs of the nested loop join on a single machine and is defined as:

$$Z := c_{dist} \cdot |DB|^2 + c_{comm} \cdot |DB| + c_{out} \cdot output_{ges} \tag{3.4}$$

Formally, the task of estimating the optimal grid is to find a partitioning $\mathcal{P}_{opt}$ that minimizes the defined costs, i.e.:

$$\mathcal{P}_{opt} = \min_{\mathcal{P}}\{cost(\mathcal{P})\} \tag{3.5}$$

3.6.2.2 Grid size determination

In the following section, we discuss the question of obtaining a good grid partitioning for the join. The complexity of the defined cost function (Equation 3.3) for a grid partitioning that is reflected in the non-linear and non-continuous dependence of all factors (communication, output size, number of distance computations) from the chosen splits make the algebraic determination of optimal cell sizes very difficult. Although there are several optimization techniques as genetic algorithm that could produce a reasonable solution, the computation of such an approximate solution would produce high additional costs.

However, intuitively, a good grid partitioning constitute cells with almost the same number of objects. In this case, dense data regions are spread and processed among multiple computing nodes. Although in reality such a high-dimensional equi-height partitioning is not optimal in terms of communication and number of computations even for uniform data distribution, it should better handle skewed data. Unfortunately, the computation of a high-dimensional equi-width grid in which the boundaries of all cells are constructed from one-dimensional splittings, is a very difficult task. The ex-

act estimation of optimal splits is already in a 2-dimensional case with only four cells probably not efficiently possible.

To simplify the problem, we, therefore, consider the one-dimensional projections only, and estimate the optimal splits based on them. More precisely, we propose to employ several value range partitioning techniques to divide the data into parts of approximately equal size. While this goal does not directly address the simultaneous optimization of the overall and maximal input/output/number distance computation from cost function (Equation 3.3), the experimental results show a significant improvement in comparison to static grids.

The usage of one-dimensional value range partitioning for data division is one of the widely used techniques in databases. For that reason, the problem of efficient computation of order statistics of large sequences has a long research history and is well studied. After the fundamental publication of Blum, Floyd, Pratt, Rivest and Tarjan [BFP+73] who showed that selection of the k-th largest element in a finite database can be done with at most $5.43N$ comparisons, several improvements of the proposed algorithm and the upper bound were introduced [MRL98, LÖ09, Tis10, Mun14]. An overview over existing techniques is given in [Pat96]. However, not every proposed algorithm is easily applicable in a parallel environment as MapReduce and therefore is not feasible for large datasets. Due to this reason, in this work, we rely on a randomized procedure from [LÖ09] (*quantile-based* strategy) and on an approach that employs equi-depth histograms (*greedy-based* strategy). In following, we describe both strategies in detail.

Quantile-based strategy Based on the technique presented in [LÖ09], to estimate m quantiles in the MapReduce framework, we employ the following extended randomized algorithm.

1. Every **mapper** draws a sample of data points of a specified size using the standard reservoir sampling technique of Vitter [Vit85].
2. From the drawn samples, every mapper determines k quantiles.
3. The k quantiles of every mapper are sent to a single reducer.

4. This **reducer** determines the desired m quantiles from the received data.

As it is often the case for randomized algorithms, the described technique produces very good approximations of real quantiles in the experimental evaluation. However, the known theoretical result for randomized determination of quantiles is, that in order to estimate the quantiles with precision $\lambda \cdot n$ with probability at least $1 - \delta$, a sample size of $\Theta\left(\frac{1}{\lambda^2} \log \frac{1}{\delta}\right)$ is needed. This means, for a database with 12 mio objects (largest DB used in our experiments) a sample size of 640.000 objects is necessary to obtain a 15000-approximation, i.e., the obtained quantiles will have a deviation of ± 15000 from the real quantiles, with probability of $1 - \delta = 0.9$. Moreover, as we compute the quantiles in a distributed way, the probability to estimate approximately correct quantiles further decreases. For that reason, we consider a second solution that employs marginal distributions to determine equi-sized partitioning.

Greedy-based strategy The greedy-based strategy pursuits the same goal as the quantile-based strategy - it tries to determine the real dimension-wise quantiles of the data. Similar to quantile-based method, it is an approximation of the real quantiles that, however, is deterministic and therefore always ends up in a solution with a guaranteed maximal fixed error that depends on the used parameters. We will shortly explain this issue in the end of this paragraph.

The basic procedure of the greedy-based strategy is to employ marginal frequency histograms with large number of bins for constructing equi-height histograms with less number of bins as in the marginal histograms. In the following, for better distinguishability, we use the word “slice” to denote the bins of the latter equi-height histogram and “bin” for bins of the precomputed frequency histogram.

As a preprocessing step, the greedy-based approach builds up marginal histograms with sufficiently small bin sizes (e.g. 1/10-th of the size of ϵ), and determines the overall number of points in the database. For a user-provided

number of desired slices per dimension, it calculates the expected number of points per slice and in a greedy way determines the slice widths. For that, it starts with the first bin and increases the size of the slice by combining subsequent bins until the expected number of points is achieved. After the first slice is found, it proceeds with other slices respectively. The result of this procedure is a histogram where in each dimension the slices have an almost equal number of objects. Certainly, this does not guarantee an equal distribution of points in the full-dimensional grid but it provides a far better approximation than equi-sized static grid partitioning.

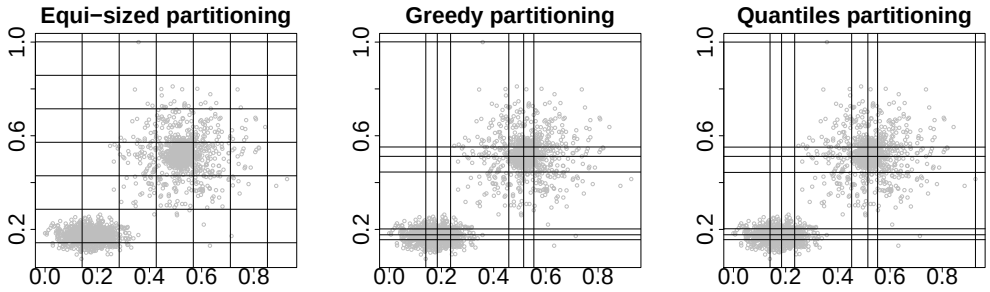
Each step of the proposed procedures is easily implemented in the MapReduce framework. The building up of a frequency histogram is a simple summation of partial histograms of every mapper, and the determination of equi-sized slices is an iterative procedure that is efficiently computable for even large number of bins in a sequential program.

The structure of the greedy-based strategy guarantees that the approximation error is at most of the size of a single slice. Therefore, for a sufficiently small slices, the approximation error will become very small.

Exemplary comparison of approaches Figure 3.15 depicts the result of the proposed techniques on a 2-dimensional point set. As one recognizes, in this simple scenario the greedy-based and quantile-based approaches perform almost equally and produce significantly better partitioning than the equi-sized static partitioning for the same number of splits per dimension. We provide a detailed evaluation of all three approaches in Section 3.6.2.4.

3.6.2.3 Incorporating variable grid sizes

As we will show in the following, the incorporating of variable grid sizes into PHiDJ only requires the adjustment of its bitcoding procedure as defined in Theorem 3.6. This solution preserves all properties of the MR-DSJ regarding filtering and efficient implementation. I.e., our PHiDJ algorithm is able to use all filter techniques and at the same time to handle variable grid sizes.



(a) Equi-sized partitioning. (b) Greedy-based partitioning. (c) Quantile-based partitioning.

Figure 3.15: Comparison of several partitioning strategies on a simple synthetic dataset. As expected, the greedy- and quantile-based approach result in an almost identical partitioning.

Theorem 3.6 *Let the domain range of every dimension be divided in differently sized bins*[§]*. Also let* bin_c^i *denote the bin position of cell* c *in dimension* i*,* $\hat{c}$ *be a home cell under consideration,* $N_\varepsilon(\hat{c})$ *be the set of neighbor cells of* $\hat{c}$ *with* $id(c) < id(\hat{c})$*, i.e., relevant cells with a distance smaller than* ε *to the* $\hat{c}$*. If for all neighbor cells* $c \in N_\varepsilon(\hat{c})$ *the bitcode* $b_1 ... b_D$ *is defined as*

$$b_i = \begin{cases} 0 & \text{if } bin_{\hat{c}}^i = bin_c^i \\ 1 & \text{if } bin_{\hat{c}}^i > bin_c^i \end{cases},$$

then performing the MR-DSJ join with provided bit coding leads to a correct and complete join result.

Theorem 3.6 states that for dealing with variable cells sizes, it is enough to check whether a cell lies in the same slice as the home cell or not and set the bit 0 or 1, respectively. An example for this bit coding is depicted in Figure 3.16 as red numbers in the cells. The upper-right green cell represents the home cell with bitcode 00. The thick-bordered rectangles represent the sets of cells with the same bitcodes 01, 10 and 11 such that the dealing with variable grid sizes does not require any additional memory for the bitcodes. As defined in Theorem 3.6, the bitcodes differ from the bitcode of the home

[§]From now on we use the shorter and more intuitive word ”bin” instead of ”slice”.

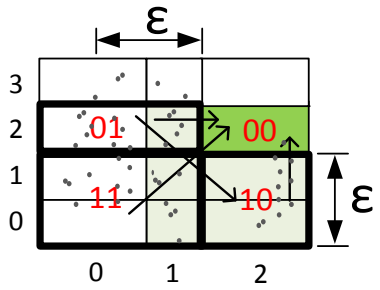


Figure 3.16: Bitcodes for variable cell sizes

cell only in the dimensions where the cells lie in different slices. The arrows between the cells sets show for which sets of cells the distances are calculated in the reducer of the home cell. All other distances will be calculated in other reducers. Figure 3.16 also shows that considering cells with varying cell sizes may lead to the case in which a home cell receives points which are further away than ε. For example, points in the left half of the cell 02 have clearly a larger distance than ε to the home cell. Nevertheless, by construction, the algorithm will not produce any duplicate pairs. Moreover, the employed filter techniques will prune most of the unnecessary calculations such that no special treatment of this problem is needed. Now we move on to the proof of Theorem 3.6. Therefore, we have to prove the completeness (Lemma 5) and the uniqueness (Lemma 6) of the solution.

For the following lemmas, let the distances $MindistCell$ and $MindestPair$ be defined as in Definitions 3.1 and 3.2, which describe the shortest distance from a point $\mathbf{p}$ to a cell c and a lower bound for the distance between two points, respectively.

Lemma 5 (Completeness) *Let $(\mathbf{p}, \mathbf{q}) \in R \bowtie_\varepsilon R$, and out_J be the output of PHiDJ, then $(\mathbf{p}, \mathbf{q}) \in out_J$, i.e., there is at least one reducer where $\mathbf{p}$ and $\mathbf{q}$ are processed.*

Proof 3.10 *Let $c_{\mathbf{p}(\mathbf{q})}$ be the cell of $\mathbf{p}$ ($\mathbf{q}$), respectively, and $\hat{c}$ be defined by its bin positions as $bin^i_{\hat{c}} = \max\{bin^i_{c_\mathbf{p}}, bin^i_{c_\mathbf{q}}\}$, for all $i \in 1 \ldots d$. Then $(\mathbf{p}, \mathbf{q})$ is processed in reducer of the cell $\hat{c}$.*

First, we prove that $\mathbf{p}, \mathbf{q}$ is sent to the reducer responsible for $\hat{c}$, i.e., we show

$mindist(c_{\mathbf{p}}, \hat{c}) \leq \varepsilon$ *and* $mindist(c_{\mathbf{q}}, \hat{c}) \leq \varepsilon$. *Here* $mindist(c_1, c_2)$ *denotes the shortest distance between the cells* c_1 *and* c_2*, i.e.,*

$$mindist(c_1, c_2) = \min_{\mathbf{p} \in c_1}\{MindistCell(\mathbf{p}, c_2)\}$$

where $\mathbf{p} \in c_1$ *denotes all corner points of the cell* c_1. *Let* $f_{1,\ldots,l}$ *be* l *dimensions in which the bin positions of* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$ *differ, i.e., where* $bin^{f_i}_{c_{\mathbf{p}}} \neq bin^{f_i}_{c_{\mathbf{q}}}$ *holds. By construction,* $\hat{c}$ *can differ only in these dimensions from* $c_{\mathbf{q}}(c_{\mathbf{p}})$. *Therefore only* $f_{1,\ldots,l}$ *contribute to* $mindist(\hat{c}, c_{\mathbf{p}(\mathbf{q})})$ *and it holds* $mindist(c_{\mathbf{p}(\mathbf{q})}, \hat{c}) \leq mindist(c_{\mathbf{p}}, c_{\mathbf{q}}) \leq \varepsilon$. *I.e.,* $\mathbf{p}, \mathbf{q}$ *are sent to* $\hat{c}$ *by some mapper.*

Now we show that $\mathbf{p}, \mathbf{q}$ *are not pruned by bitcode pruning. Therefore, consider* $\hat{c}$ *as home cell and check the bitcodes of* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$.

Case 1: $bin^i_{c_{\mathbf{p}}} = bin^i_{c_{\mathbf{q}}}$**.** *In this case both* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$ *have* 0 *at position* i *in the bitcode. By construction of the bit pruning* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$ *will not be pruned.*

Case 2: $bin^i_{c_{\mathbf{p}}} \neq bin^i_{c_{\mathbf{q}}}$. *Either the bitcode of* $c_{\mathbf{p}}$ *or the bitcode of* $c_{\mathbf{q}}$ *at position* i *has value* 1 *but not both at the same time, since due to* max*-function only* $c_{\mathbf{p}}$ *or* $c_{\mathbf{q}}$ *can differ from* $\hat{c}$ *in this dimension. By construction of the bit pruning* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$ *are not pruned. Overall,* $\mathbf{p}, \mathbf{q}$ *are not pruned and are processed in cell* $\hat{c}$. ■

Lemma 6 (Uniqueness) *For all cells* c *with* $\mathbf{p}, \mathbf{q} \in c$ *(i.e.,* $\mathbf{p}, \mathbf{q}$ *were sent to* c*) it holds:* $c = \hat{c}$ *or the computations of distances between the cells of* $\mathbf{p}$ *and* $\mathbf{q}$ *are pruned.*

Proof 3.11 *Let* $c \neq \hat{c}$*, with* $\hat{c}$ *being constructed as in the proof of Lemma 5, i.e. it must hold* $mindist(c, c_{\mathbf{q}}) \leq \varepsilon$ *and* $mindist(c, c_{\mathbf{p}}) \leq \varepsilon$ *but there is at least one dimension* i *where* $bin^i_c \neq \max\{bin^i_{c_{\mathbf{p}}}, bin^i_{c_{\mathbf{q}}}\}$. *I.e., as well* $bin_{c_{\mathbf{p}}}$ *as* $bin_{c_{\mathbf{q}}}$ *deviate from* bin_c *in dimension* i. *Therefore* $bin^i_c \neq bin^i_{c_{\mathbf{p}}}$ *and* $bin^i_c \neq bin^i_{c_{\mathbf{q}}}$. *It follows, that* $bitcode(c^i_{\mathbf{p}}) = 1$ *and* $bitcode(c^i_{\mathbf{q}}) = 1$ *such that* $c_{\mathbf{p}}$ *and* $c_{\mathbf{q}}$ *are pruned by the bitcode pruning in every cell except for* $\hat{c}$. ■

The proof of Theorem 3.6 now directly follows from Lemma 5 and 6.
Last, we prove that using variable grid sizes does not affect the filters introduced in MR-DSJ (Theorem 3.7).

Theorem 3.7 *Let c be a cell and $\mathbf{p}$, $\mathbf{q}$ be two points in the database. Then it holds:*

1. *Both, reducer-side and mapper-side $MindistCell(\mathbf{q}, c)$ filters do not exclude any result pairs $(\mathbf{p}, \mathbf{q}) \in R \bowtie_\varepsilon R$*
2. *$MindistPair(\mathbf{p}, \mathbf{q}) > \varepsilon \Rightarrow dist(\mathbf{p}, \mathbf{q}) > \varepsilon$, i.e., if $MindistPair(\mathbf{p}, \mathbf{q})$ filters a pair $(\mathbf{p}, \mathbf{q})$ then $(\mathbf{p}, \mathbf{q}) \notin R \bowtie_\varepsilon R$.*

Proof 3.12 *Both, the proof for reducer- and mapper-side $MindistCell(\mathbf{q}, \mathbf{c})$ filters are almost identical to the one for MR-DSJ. For that reason, we only provide an intuition for both proofs here. First we consider the reducer-side $MindistCell(\mathbf{q}, \mathbf{c})$ filter. Let $\mathbf{q}$ be a point in some cell c and $\hat{c}$ be a home cell. By construction of the $MindistCell$ filter, each dimension in which c does not lie in the same slice as $\hat{c}$ contributes to the distance. At the same time, if two cells deviate from the home cell in the same dimension, they are pruned by the bitcode pruning. This results in the fact that the distance of $\mathbf{q}$ to the home cell is minimal, i.e., there is no other non-pruned cell to which $\mathbf{q}$ is nearer. Therefore, if $\mathbf{q}$ is pruned by the $MindistCell$ filter in the reducer, no result pairs are lost. For the mapper-side pruning one can use similar arguments to prove the completeness of the filter.*

The completeness of the $MindistPair(\mathbf{p}, \mathbf{q})$ filter in PHiDJ results from the fact that $MindistPair$ is defined on the points $\mathbf{p}, \mathbf{q}$ and the home cell. I.e., the cells of $\mathbf{p}$ and $\mathbf{q}$ are not considered explicitly at all and therefore we can use the results from MR-DSJ without any changes. ■

Theorem 3.6 and 3.7 prove that PHiDJ is a correct similarity self-join algorithm which exploits all filter techniques of MR-DSJ.

3.6.2.4 Experimental evaluation of variable grids quality

In this part, we experimentally evaluate the quality of the variable grids in comparison to static grids in terms of partitioning costs as defined in Equation 3.3. For that, we apply the PHiDJ algorithm with

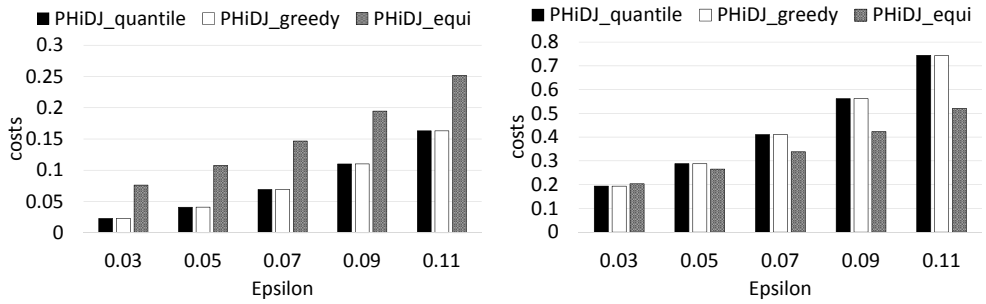
1. *greedy*- and *quantile-based* strategies for cell size estimation, and

2. the *static grid*

on subsets of synthetic data of different dimensionality (2D to 20D), and on a real world Flickr datasets. A detailed description of the datasets is provided in Sections 3.7.1 and 3.7.2. We investigate the change of costs for several ε values and multiple number of bins (that corresponds to different sizes of the cells) for several database sizes. For these experiments, we normalize all dimensions of the databases to the $[0, 1]$ range for making the comparison of several databases easier. For the upcoming experiments, we set the costs for the communication of a single object, for a single distance computation and for output costs as:

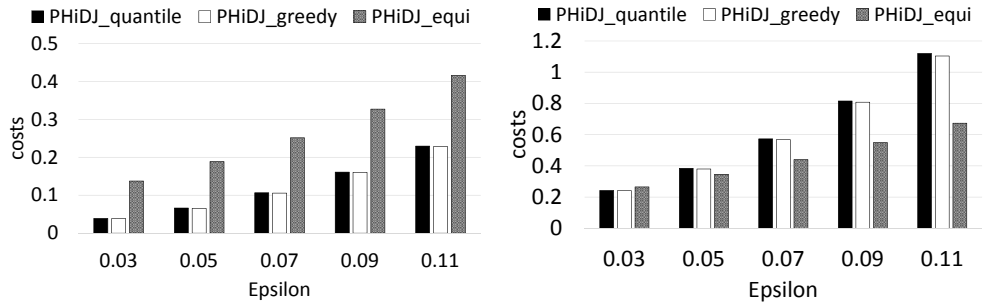
- $c_{dist} = 1.6872 \times 10^{-6} \cdot d$: Time to compute a single squared Euclidean distance for d-dimensional objects
- $c_{comm} = \frac{8(bit) \cdot 8(byte) \cdot d}{1 \times 10^5}$: Time to communicate a single d-dimensional double-object in the "fast" network.
- $c_{comm}^{slow} = \frac{8(bit) \cdot 8(byte) \cdot d}{1 \times 10^3}$: Time to communicate a single d-dimensional double-object in a "slow" network. We use this value to show the effect of the network speed on the grid costs.
- $c_{out} = \frac{2 \cdot 8(byte) \cdot d}{100 \cdot 1024 \cdot 1024}$: Time to write a result pair with 2 double-objects using a 100 MB/sec fast hard drive

Influence of the ϵ-range parameter In the following experiments, we compare the results of the join with proposed partitioning strategies for a constant number of bins=7 on 10^5 large subsets of the synthetic 10-dimensional and the (also 10 dimensional) Flickr datasets. The results on other datasets show a similar picture and are omitted here. Figures 3.17 and 3.18 depict the join costs for the proposed partitioning strategies on the synthetic and Flickr dataset using the fast and the slow network, respectively. The abbreviations *PHiDJ_quantile*, *PHiDJ_greedy* and *PHiDJ_equi* stand for the quantile-based, greedy-based and equi-sized (static) partitioning strategies, respectively.



(a) Costs, fast network. (b) Costs, slow network.

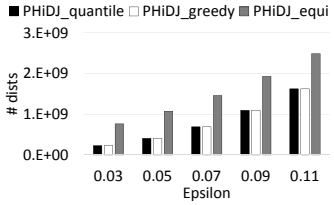
Figure 3.17: Costs for varying ε parameter on the synthetic 10D dataset with 100k objects.



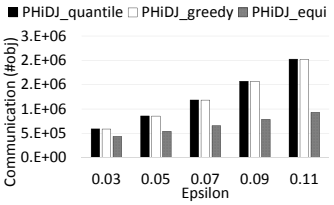
(a) Costs, fast network. (b) Costs, slow network.

Figure 3.18: Costs for varying ε parameter on the Flickr dataset with 100k objects.

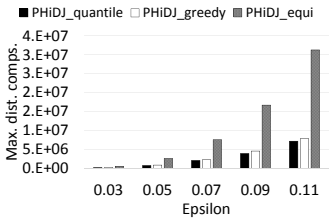
Further, Figures 3.19 and 3.20 show the measured overall and maximal number of computations, overall and maximal communication, and maximal output size for synthetic and Flickr datasets. For both datasets, we observe similar effects. The quantile-based and greedy-based approaches show a significantly better workload balancing than the equi-sized grid. As well maximal number of distance computations (Figures 3.19(c), 3.20(c)) as maximal input (Figures 3.19(d), 3.20(d)) and maximal output (Figures 3.19(e), 3.20(e)) are multiple times lower for the both variable partitioning techniques in comparison to the static approach. However, at the same time, the overall communication (Figures 3.19(b), 3.20(b)) of the quantile-based and greedy-based approaches is higher for all ε values and it is growing at a



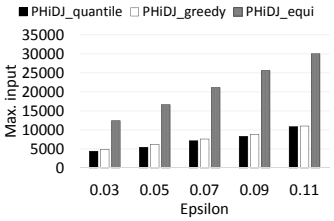
(a) Overall distance computations.



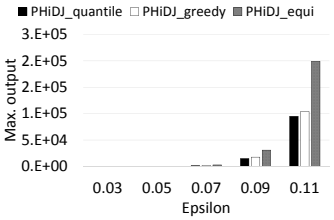
(b) Overall communication.



(c) Maximal number of distance computations of a single reducer.



(d) Maximal input (=communication) of a single reducer.

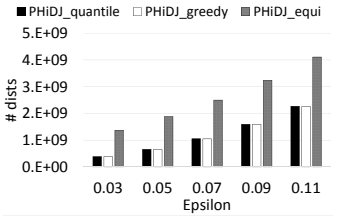


(e) Maximal output of a single reducer.

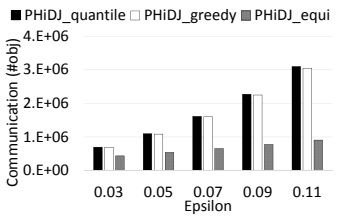
Figure 3.19: Influence of ε parameter on the synthetic 10D dataset with 100k objects.

similar or higher rate as the communication of the equi-sized grid approach. To understand the reasons for this effect, recall that to compute the join result, each point in the MR-DSJ and PHiDJ algorithms is distributed to all cells within ε radius of its home cell, despite the ones pruned by bitcode filter. As the quantile-based and greedy-based approaches adapt the cell sizes to the data distribution, the sizes of several cells may become smaller than ε that in its turn results in a higher replication of objects from this and its neighboring cells.

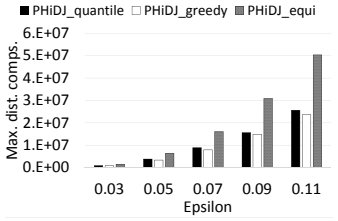
Hence, the adaptation of the grid to the data distribution has two opposite effects on the overall join costs. The better load distribution mirrored by the smaller number of overall distance computations and lower maximal reducer output is paid by a larger overall communication amount such that the question which strategy to use mainly depends on the employed hardware. By adjusting the constants of the cost function, we simulate the effect of arbitrary fast network speeds. We depict the costs of the similarity self-join for



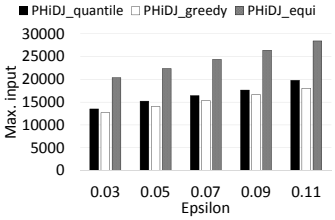
(a) Overall distance computations.



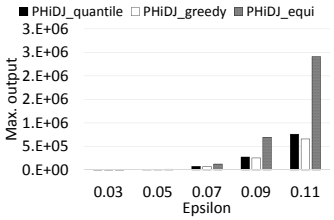
(b) Overall communication.



(c) Maximal number of distance computations of a single reducer.



(d) Maximal input (=communication) of a single reducer.



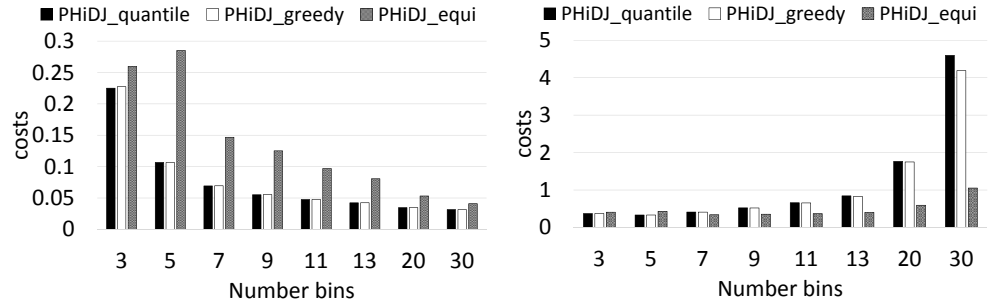
(e) Maximal output of a single reducer.

Figure 3.20: Influence of ε parameter on the Flickr dataset with 100k objects.

fast and slow networks for the synthetic and Flickr datasets in Figures 3.17 and 3.18. As expected, the PHiDJ-equi approach performs better in cluster with a slow network, while the both variable grid approaches benefit from a fast network.

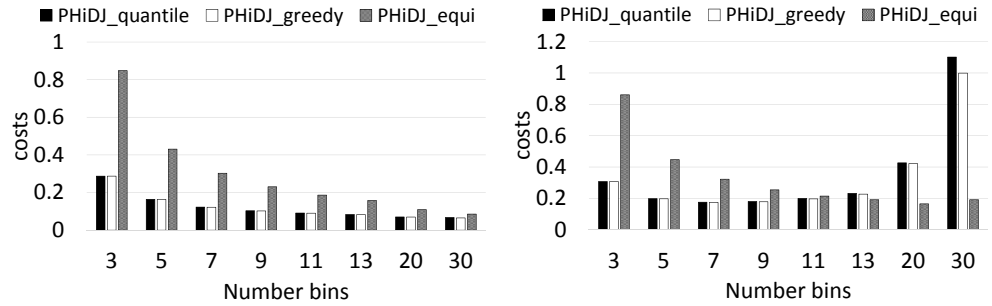
Influence of the bin number per dimension In this section, we investigate the influence of the varying bin numbers per dimension on the costs of the grids. We compare the quantile-based, greedy-based and static grids on 10^5 large samples of the Flickr and synthetic 10-dimensional datasets for $\varepsilon = 0.07$. Figures 3.21 and Figures 3.22 depict the obtained costs in a slow and fast network for both datasets.

The values for individual cost components are depicted in Figures 3.23 and 3.24 for the synthetic and Flickr datasets, respectively. The observed result confirm the observations from the previous paragraph. Again, the quantile- and greedy-based approaches perform equally well and show the better capability of variable grid to balance the workload of the join on the



(a) Costs, fast network. (b) Costs, slow network.

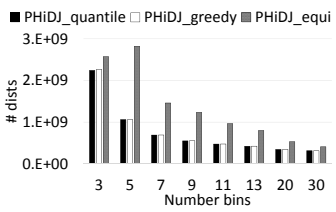
Figure 3.21: Costs for varying number of bins on the synthetic 10D dataset with 100k objects.



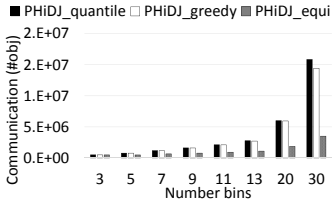
(a) Costs, fast network. (b) Costs, slow network.

Figure 3.22: Costs for varying number of bins on the Flickr dataset with 100k objects.

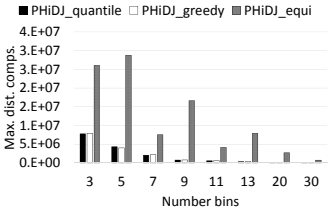
multiple computing nodes. It mirrors in a less number of overall number of distance computation, lower number of maximal number of distance computation and lower maximal input / output of the single reducer for a large range of used bins. However, similar to the experiments from the previous paragraph, the advantage through the better workload balancing is weakened by a significantly higher overall communication that, moreover, shows a much larger growth rate for increasing number of bins. Furthermore, the positive effects of variable bins sizes vanishes for growing bin numbers. The latter effect is easily explainable, as growing number of bins results in smaller cell sizes and therefore smaller relative differences between static equi-sized and variable approaches when the maximal input / output and therefore



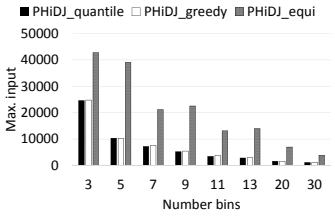
(a) Overall distance computations.



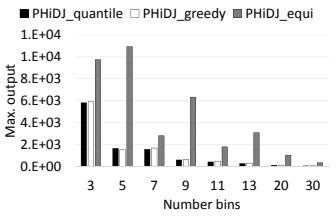
(b) Overall communication.



(c) Maximal number of distance computations of a single reducer.



(d) Maximal input (=communication) of a single reducer.



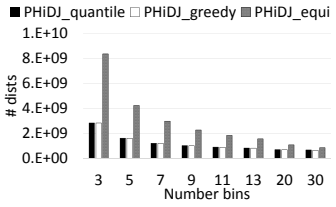
(e) Maximal output of a single reducer.

Figure 3.23: Influence of the number of bins on the synthetic 10D dataset with 100k objects.

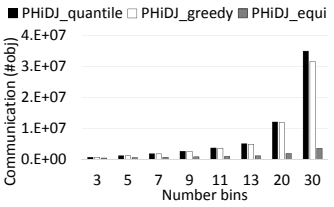
overall number of computations are considered.

In Figures 3.21 and 3.22, we show the effect of different network speed on the join costs and similar to experiments from the previous paragraph one observes a significant effect on the join performance.

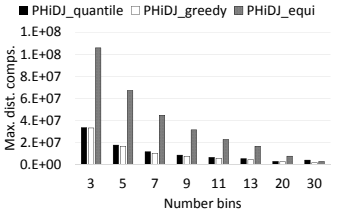
Despite the significant communication increase, the quantile-based and greedy-based approaches perform equally well or better than the static approach for quite a large number of bins. Considering the fact that for high dimensional data the number of reducers growths (potentially) exponentially with the number of bins per dimension (as there are exponentially many cells in the grid), one will usually use a small number of bins per dimension and that makes the quantile-based and greedy-based approaches considerably better choices for grid partitioning.



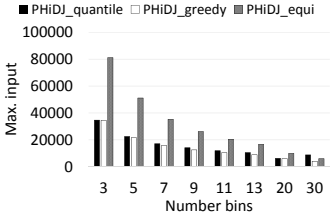
(a) Overall distance computations.



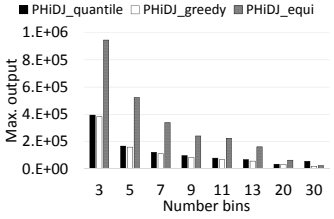
(b) Overall communication.



(c) Maximal number of distance computations of a single reducer.



(d) Maximal input (=communication) of a single reducer.



(e) Maximal output of a single reducer.

Figure 3.24: Influence of the number of bins on the Flickr dataset with 100k objects.

3.6.2.5 Evaluation conclusion

The experimental evaluation of the proposed strategies for choosing the cell sizes shows a diverse picture. On the one hand, the greedy- and quantile-based approaches clearly outperform the static grid approach when only the maximal workload of a single reducer is considered. From that point of view, both strategies achieve the desired goal of the workload balancing task. On the other hand, the better adaptation of the cell sizes to the data distribution leads to a higher overall communication, as the cell sizes may become smaller than ε that then leads to a higher data replication. At the same time, a better adaptation of the grid also leads to the less number of overall distance computations and lower maximal output that positively influence the performance of the join algorithm.

Concluding, the choice of the best partitioning strategy is a non-trivial task and requires a precise heuristic for communication amount and number of computations. Due to the complexity of such a heuristic (that would

Table 3.2: Global variables of the program.

ε_k	Reduced query radius (as defined earlier)
gridWidth	Widths of the grid per dimension
dgIndices	Indices of dimensions in certain dimension groups
dist()	L_m-based distance function
$dist_{dg}()$	Distance function restricted to a dimension group dg
getCellId()	Returns the bin ID of the cell in a certain dimension
size()	Returns the size of an array object

depend on the data distribution knowledge and algorithm structure), the widely used approach to use the costs of a small sampling as approximation for the real join execution costs could be a worth of considering alternative.

3.6.3 Implementation in MapReduce

In this section, we provide a pseudo-code implementation of the PHiDJ algorithm. The code of the algorithm consists of the map and reduce functions presented in Listings 3.3 and 3.4, respectively. Both, mapper and reducer rely on the global variables and functions described in Table 3.2. Additionally, in the mapper, two further arrays $distToCells[][]$ and $upperDist[]$ are used. $distToCells[dim][sliceID]$ stores the distance from the origin to the farther side of a cell in the slice $sliceID$ in dimension dim. I.e., $distToCells$ stores the coordinates of the farthest corners of the cells. $upperDist[dim]$ contains the distance of a point to its next neighbor cells in dimension dim.

Listing 3.3: Map function for PHiDJ

```
1 upperDist[] = null
2 distToCells [][] = dimension−wise upper bound distances to the cells // cf. text
3
4 (outkey, outvalue): Intermediate result pair
5 void map(int id, float[] coord)
6     outvalue.coords = coords
7     // SSJ for all dimension groups with reduced epsilon
8     for (curDg=1; curDg <= numDg; curDg++)
9       int numDimsInDg = size(dgIndices[curDg])
```

```
10      outkey.dimGroup = curDg
11      outkey.cellID  = new short[numDimsInDg]
12      upperDist = new double[numDimsInDg]
13
14      // Compute cellID and upperDist for current dim. group
15      for (byte i = 0; i < size(dgIndices[curDg]); i++) // for each dim. group
16        // current (real) dimension in dimension group
17        byte subDim = dgIndices[curDg][i]
18        // get ID of the cell in the dimension sumDim
19        short pntCellId=getCellId(subDim, coords[subDim])
20        outkey.cellID[i]=pntCellId
21        upperDist[i] = distToCells[subDim][pntCellId] − coords[subDim]
22
23      mapRecursive(0, 0, 0.0)
24
25  void mapRecursive(byte subDimIdx, int bitcode, double minDistP)
26  // process all dimensions in the dimension group
27    if (subDimIdx < size(dgIndices[curDg]))
28      // Send to all neighboring reducers with the same cellID [subDimIdx]
29      mapRecursive(subDimIdx + 1, bitcode << 1, minDistM)
30
31      // Send to all reducers with bigger cellID [subDimIdx] & in range w.r.t. ε_k^m
32      byte subDim = dgIndices[curDg][subDimIdx]
33      short cellID_temp = outkey.cellID[subDimIdx]
34      short c=0 // number cells with distance < ε_k in subDim
35      short hs=outkey.cellID[subDimIdx] // cell index of point
36
37      for(float d=upperDist[subDimIdx], newMinDistM=minDistM+d^m;
38            (newMinDistM ≤ ε_k^m) && // Mapper side pruning
39            hs+c+1 < size(gridWidths[subDim]); // check grid bounds
40              d += gridWidths[subDim][hs+c+1], // accumulate distance
41              newMinDistM = minDistP + d^m, c++)
42        outkey.cellID[subDimIdx]++ // ID of neighbor cell
43        mapRecursive(subDimIdx + 1, (bitcode << 1) | 1, newMinDistM)
44
```

```
45      outkey.cellID[subDimIdx] = cellID_temp
46    else // all indices are set, emit the pair
47      outvalue.minDistM = minDistM
48      outkey.bitcode = bitcode
49      emit(outkey, outvalue)
```

The map-function first precalculates the cell ID for each incoming point $pt = (id, coords)$ and the distances of pt to the borders of the cell pt lies in. These values are stored in the $outkey.cellID$ and $upperDist$ variables, respectively, and are used later on for pruning distance calculations. Hereby, $outkey$ denotes the key of the emitted intermediate pair for a point pt and consists of 1) the cell ID where pt will be sent to, 2) the dimension group and 3) the bitcode of pt. After initial precomputations, the *mapRecursive* function is called and recursively generates the ids outkey.cellID of all cells with a distance smaller than ε_k to pt and emits the intermediate pairs (outkey, outvalue) to those cells. Therefore, in the for-loop the distance $newMinDistM$ between the point pt (line 37) and the cell under consideration is recursively computed in every dimension of the current dimension group . Here, the $gridWidth$ variable contains the cell widths of the grid in every dimension. I.e., the exchange of the values of this variable leads to a different partitioning of the grid. The dimensions of the dimension group are stored in the variable $dgIndices[curDg]$, where $curDg$ denotes the current dimension group. In the same for-loop, we accumulate the m^{th} powers of the mindist-distance of pt to the respective cells and use this value for implementing the $MindistCell$-filter by the condition in line 38. I.e., an object pt is only then sent to a cell if the $MindistCell$ does not exceed the threshold ε_k. For known cell sizes in each dimension (e.g. after using a preprocessing step for setting variable grid sizes), an additional check (line 39) guarantees that no points are emitted to cells outside the grid. As the number of border cells grows exponentially in the number of dimensions, this can significantly decrease the data replication. The bitcode $bitcode$ of every point pt is calculated in the same recursion. Since $bitcode$ in every dimension is only set to 1 if the object lies in another slice as the home cell, the calculated bitcodes correspond to

the bitcode structure defined in Theorem 3.6. The result of the mapper is a set of intermediate pairs (outkey,outvalue), where each point pt correspond to as many pairs as there are cells in all dimension groups to which pt is sent. As mentioned earlier, the $outkey$ consists of the cell ID to which pt is sent to, the dimension group and the bitcode. Therefore, a single reducer receives all the points from a single cell of a single dimension group and all cells in the ε_k radius with smaller ids. The $outvalue$ comprises the bitcode, $MindistCell$ to the home cell($minDistM$), the ID and the full dimensional coordinates of the point pt. Summarized, the mapper of PHiDJ extends the MR-DSJ mapper by the capability to use dimension groups and variable grid widths. At the same time, it preserves all features of MR-DSJ, i.e., a simple and efficient implementation and the mapper-side $MindistCell$ filter.

Now we proceed with the description of the reduce function, which is given in Listing 3.4. As was shown in Theorem 3.6, a simple redefinition of the bitcode function enables the original MR-DSJ algorithm to compute the join result in a grid with variable cell sizes. The provided implementation of the mapper in Listing 3.3 performs exactly this task. The reduce function mostly remains unchanged in comparison to the original MR-DSJ algorithm. Due to the employed secondary sort (cf. Section 3.3.3, page 45), the incoming points are sorted in ascending *bitcode* order and arrive grouped by their *cell ID*. The points of a single cell are stored in a buffer ($valBuffer$) and each buffer is assigned an ID equal to the bitcode of the cell. Like in the MR-DSJ algorithm, the bitcode filter (line 8) is used in order to decide for which cells the distances of the contained points must be calculated. As was shown in MR-DSJ, this is the case for the cells that do not share any from the home cell deviating dimension. After the bitcode pruning, the $mindistCell$ filter, (line 9) and the $mindistPair$ filter (line 12) conditions are checked. The needed values are provided from the mapper phase and are stored in the intermediate values. If all conditions are fulfilled, an exact distance calculation is performed. The main difference arises from the fact that in PHiDJ a reducer has to calculate result pairs of a single dimension group dg and not in the full dimensional space. Therefore, all conditions and distance computations are restricted to the current dimension group dg (denoted by $dist_{dg}$ and

Listing 3.4: Reduce function for PHiDJ

```
1  void reducer(key, Iterator values)
2      bitcodeBuffer.clear()
3      foreach val in values
4        int bc = 0 // buffer bitcode
5        // Value in bitcodeBuffer are in ascending bitcode order
6        foreach valBuffer in bitcodeBuffer
7          if(val.bitcode + bc ≥ (maxFlag << 1)) break
8          if((val.bitcode & bc) == 0) // Bitcode filter
9            if (minDistM(val, bc) ≤ ε_k^m) // mindistCell filter
10             foreach otherVal in valBuffer
11               // mindistPair filter
12               if(val.minDistM + otherVal.minDistM ≤ ε_k^m)
13                 if(dist_dg(val, otherVal)^m ≤ ε_k^m ) // Candidate found?
14                   // Previously emitted?
15                   if(!alreadyEmitted(val, otherVal))
16                     if(dist(val, otherVal)^m ≤ ε^m) // Found match
17                     emit(val, otherVal)
18                     emit(otherVal, val) // if desired
19         bc++
20       if (val.bitcode < maxFlag) // relies on secondary sort
21         bitcodeBuffer[val.bitcode].add(val)
22
23       emit(val,val) // reflexive pair
```

ε_k). As described earlier, for preventing duplicate results for each computed candidate pair (x, y), it must be checked whether it was already emitted in another dimension group. This check is performed in line 15. In the function "alreadyEmitted", the distances between the objects val and $otherVal$ for all dimension groups having a smaller order number as the currently proceeding dimension group dg are computed. As stated earlier, we assume that there is a predefined ordering of the dimension groups. Only if there are no further dimension groups in which $(val, otherVal)$ is emitted, the current reducer outputs the result tuple $(val, otherVal)$.

3.6.4 Analysis

In this section, we provide an analytical evaluation of the PHiDJ algorithm. The main result will be that the communication of PHiDJ decreases exponentially with the number k of dimension groups at the cost of an exponential growth of computations in comparison to the MR-DSJ algorithm.

First, we assume an equi-sized grid with a cell size equal to ε and a d-dimensional uniform data distribution in the $[0,1]^d$ domain and consider grids with variable cell sizes later. Let the number of elements in the database be $|DB|$. We assume that the d dimensions are split in k equi-sized dimension groups $D_1, \cdots, D_k$, i.e., $|D_i| = \frac{d}{k}$ for all D_i. As distance function we use the Minkowsky norm L_m, i.e., ε_k is then defined as in Theorem 3.5, i.e., $\varepsilon_k = \frac{\varepsilon}{\sqrt[m]{k}}$.

First, we consider a single dimension group D_i. Since for equi-sized grids and a single dimension group, PHiDJ is almost equal to MR-DSJ, one only has to adjust the size of the cells to ε_k and then the results from the analysis part of Section 3.3.1 can be used. The number of points per cell in D_i therefore equals $\varepsilon_k^{D_i} \cdot |DB|$; the communication in D_i is $comm_{D_i} = 2^{|D_i|} \cdot |DB|$, and the number of distance computations equals $comp_{D_i} = (3\varepsilon_k)^{|D_i|} \cdot |DB|^2 = k^{-\frac{d}{m \cdot k}} \cdot (3\varepsilon)^{|D_i|} \cdot |DB|^2$. With respect to all dimension groups, we have $comm_{total} = k \cdot 2^{|D|} \cdot |DB|$ and $comp = k^{1-\frac{d}{m \cdot k}} \cdot (3\varepsilon)^{D_i} \cdot |DB|^2$.

In order to prevent duplicates, for each candidate pair $(\mathbf{p}, \mathbf{q})$ in a dimension group D_i it has to be checked whether $(\mathbf{p}, \mathbf{q})$ was already found in one of the previous dimension groups D_j, $j < i$. We define the random variable X as the total number of dimension groups which have to be additionally checked. Assuming the uniform distribution $P(X = j) = \frac{1}{k}$ for all $j = 1 \ldots k$, the expected number of additional calculations is $\mathbb{E}(X) = \sum_{i=1}^{k} i\frac{1}{k} = \frac{1}{2}(k+1)$. Thus, the total number of computations is

$$comp_{total} = \frac{1}{2} \cdot (k+1) \cdot comp = (k+1) \cdot k^{1-\frac{d}{m \cdot k}} \cdot \frac{1}{2} \cdot (3\varepsilon)^{|D|} \cdot |DB|^2.$$

Comparison to MR-DSJ In comparison to the MR-DSJ algorithm, PHiDJ performs $\frac{comm_{total}}{comm_{MR-DSJ}} = k \cdot 2^{d(\frac{1}{k}-1)}$ times less communication and $\frac{comp_{total}}{comp_{MR-DSJ}} = (k+1) \cdot k^{1-\frac{d}{m \cdot k}}$ times more computations. I.e., the exponential decrease of

the communication results in an exponential increase of the distance computations. Thus, PHiDJ achieves its efficiency by exploiting the fast CPUs and relatively slow network speed of today's hardware. A shift to a much faster network architecture will significantly decrease the advantage of PHiDJ in comparison to MR-DSJ.

Analysis of fixed, dimension-dependent cell sizes Now we consider how dimension-wise different cell sizes (inside a dimension the cell sizes are still fixed) influence the communication and the number of computations in a single dimension group.

Communication: Let c be a cell with bin IDs $(c_1, \cdots, c_d)$ and $NC(c)$ its neighbor cells. Due to the bitcode pruning, each point from c is sent to all cells inside the ε radius that have a greater ID than c. Let n_i denote the number of cells with bin IDs equal to i (i.e., the cells lying in the same slice as c in dimension i) that "survive" the bitcode pruning. In this case, $NC(c)$ is upper bounded by $NC(c) \leq \prod_{i \in \{1,\cdots d\}} n_i$, i.e., the increase of neighbor cells in a single dimensions, which corresponds to the decrease of cell width in this dimension, contribute multiplicatively to the overall number of neighbor cells. Please note, that depending on the used distance function this upper bound is more or less tight. While for L_1 or L_2 norms, the real number of neighbors can be significantly lower, for the L_∞ the equality $NC(c) = \prod_{i \in \{1,\cdots d\}} n_i$ holds. The overall communication $\#comm$ is, therefore, upper bounded by:

$$\#comm = \sum_{c \in \mathcal{C}} NC(c) \cdot p(c),$$

where $\mathcal{C}$ is the set of all cells in a grid, and $p : \mathcal{C} \to \mathbb{N}^0$ is the distribution function assigning the number of points to each cell. Therefore, the overall communication depends on the underlying data distribution; for uniform data and L_∞ norm, however, the decrease the cells sizes will result in an exponential increase of communication.

Number of computations of a single reducer: Since for the exact solution there is probably no closed solution, we provide the upper and lower bounds for this quantity. Let us consider the case where each cell has equal

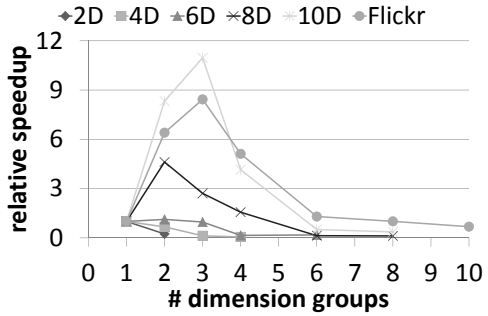
widths $\delta = \frac{\varepsilon}{n}$ in every dimension, where $n \in \mathbb{N}$ is the number of cells within ε radius from the home cell. Then, if we assume a uniform data distribution, the volume of every cell is equal to δ^d, and for all cells c whose points are sent to a home cell $\hat{c}$ it holds that $mindist(c, \hat{c}) \leq n \cdot \delta$, i.e., the furthest neighbor cell c of a home cell $\hat{c}$ is $n-1$ cells away (or n, if the home cell is considered, too). For a uniform data distribution, the volume of a cell c corresponds to the fraction of objects in the cell. The number of computations performed by a single reducer equals:

$$\#comps = \sum_{i=1}^{d} \binom{d}{i} (n\delta^d)^i \cdot \sum_{j=1}^{d-i} \binom{d-i}{j} (n\delta^d)^j + H,$$

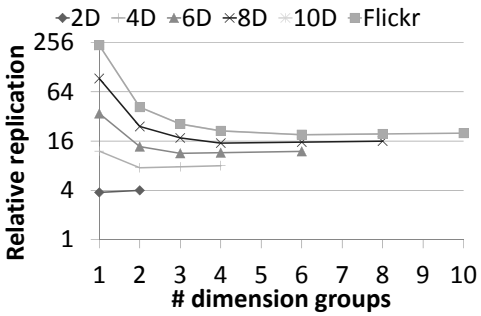
with $H = \delta^d \cdot ((n\delta^d)^d - \delta^d)$. Here, the first sum and the binomial coefficient give the amount of numbers with i bits set to 1. The second sum is the amount of numbers with $d-i$ bits set to 1, i.e., the first sum counts over all existing bitcodes for a d-dimensional data space and the second sum gives for every cell with i ones in the bitcode the number of cells which will not be pruned by the bitcode pruning. The value $(n\delta^d)^j$ is the volume of the group of cells with the same bitcode. Thus, this part of the formula calculates the amount of calculations not involving the home cell. All calculations with the home cell are gathered in H. Here we calculate the number of calculations from all cells of one reducer without the home cell ($(n\delta^d)^d - \delta^d$) to all points in the home cell δ^d. The above term can be simplified to:

$$\#comps = 1 - (\gamma+1)^{d-1} - (\gamma+1)^d + (2\gamma+1)^{d-1} \cdot (\gamma+1) + H$$

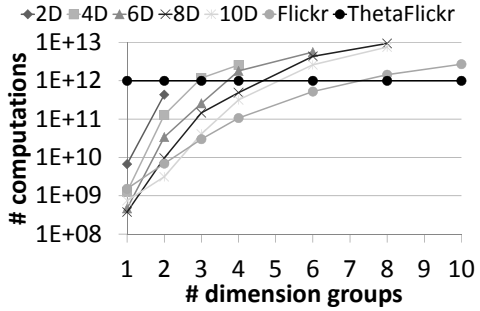
where $\gamma = (n\delta^d) = \left(n(\frac{\varepsilon}{n})^d\right)$. We observe that changing the size of the cell has an exponential influence on the number of computations inside a single reducer. The upper and lower bounds for the real number of calculations result from setting δ to the maximal or minimal size of a cell in all dimensions, respectively.



(a) Relative speedup



(b) Relative replication



(c) Distance computations

Figure 3.25: Experiments on synthetic datasets with increasing dimension group numbers (using PHiDJ-static)

3.7 Experiments

In this section, we summarize the performance evaluation results for PHiDJ. We denote by PHiDJ-static the version of the algorithm where in each dimension the cells are still fixed, and by PHiDJ-var the version where for each dimension the grid sizes are fitted by the *greedy-based partitioning* to the data distribution as explained in Section 3.6.2. In our experiments, we evaluate the performance of both versions. We restrict the evaluation to the greedy-based approach due to two reasons. First, the results in Section 3.6.2.4 show an equally good and in several cases better adaptability of the greedy-base partitioning in comparison to the quantile-based partitioning. Second, as was stated in Section 3.6.2.2, the quality of the randomized determination of quantiles may become low for a small sampling size. At the same time, the greedy approach will constantly provide an in the worst case $\frac{|DB|}{\#\text{number bins}}$-

approximation of the quantiles. Using a sufficiently large number of bins (e.g. 1000 or 10000), this provides a very good approximation for the database sizes used in our experiments.
Section 3.7.1 covers the experiments with synthetically generated datasets ranging from 2 to 20 dimensions, and Section 3.7.2 refers to a 10 dimensional real dataset generated from the 2010-MIRFLICKR-1M[¶] database by means of the Signature Quadratic Form Distance [BUS10]. Section 3.7.3 deals with the influence of the grid width on the performance on 2 and 10-dimensional data. In Section 3.7.4, we investigate the influence of variable grid sizes. The experiments were performed on the same hardware as described in 3.4. Besides the MR-DSJ algorithm, we also compare our approach with the θ-join approach from [OR11] and the MR-SimJoin approach from [SR12].

We use three main performance measures for our evaluation: The first one is the relative runtime speedup with respect to the runtime using only one dimension group (in Sections 3.7.1 and 3.7.2) or to the runtime using the original grid width ε (in Section 3.7.3). Using only one dimension group is the special case where one always considers all dimensions for the distance computations instead of dimension subsets. This case is equivalent to the MR-DSJ approach. The second measure is the relative/absolute replication factor, which is the number of reducers each point is sent to. It serves as an indirect measure for the communication cost and the degree of parallelism. The higher replication factor implies a higher communication cost and at the same time a higher amount of parallel performed distance computations.

The last measure is the total number of distance computations performed in all reducers consisting of the number of computations in the projected subspace of the current dimension group, the computations with respect to all dimensions, and the ones performed for the reducer-side pruning techniques.

[¶] `http://press.liacs.nl/mirflickr`

3.7.1 Synthetic data

The synthetic datasets for the dimension group evaluation consist of 3.1 million points each. They were generated according to two normal distributions with uncorrelated dimensions and a standard deviation of 2 in each dimension. Their means were chosen to have an Euclidean distance of at least 10 ($= 2 \cdot 2.5 \cdot stddev$) to each other so that the distributions are separated and do not "overlap" much. One million points were generated with respect to each distribution and in addition to that an uniformly distributed noise of 100 thousand points was added. The area for the uniform distribution was defined as a cuboid covering 99% of all the normally distributed points. Figure 3.26 depicts the marginal distribution of the 10-dimensional synthetic dataset. Such datasets were generated for 2, 4, 6, 8, 10, 15 and 20 dimensions. The applied ε_ds[‖] for d dimensions are $\varepsilon_2 = 0.1$, $\varepsilon_4 = 0.32$, $\varepsilon_6 = 0.46$, $\varepsilon_8 = 0.56$, $\varepsilon_{10} = 0.63$, $\varepsilon_{15} = 0.74$ and $\varepsilon_{20} = 0.79$.

In these experiments, we only consider the PHiDJ-static variant. The results for the 2 to 10-dimensional datasets are summarized in the Figures 3.25(a) to 3.25(c), which depict the runtime, the replication factor (with a logarithmic scale on the y-axis) and the total number of computations for each dimension group setting. Three effects can be seen here:

1. Except for the 2-dimensional data, the replication factor drops drastically even for low dimension group settings.

2. Except for the 2-dimensional data, the number of distance computations only increases insignificantly for low dimension group settings.

3. These two effects become even stronger the higher the dimensionality of the data gets.

This is confirmed by the relative runtime speedup summary in Figure 3.25(a). Here, the saved communication costs begin to compensate for the additional computation costs for datasets of at least 6 dimensions and the best results are always achieved for relatively low dimension group counts. The latter

[‖] Not to be confused with reduced ε_k in previous sections.

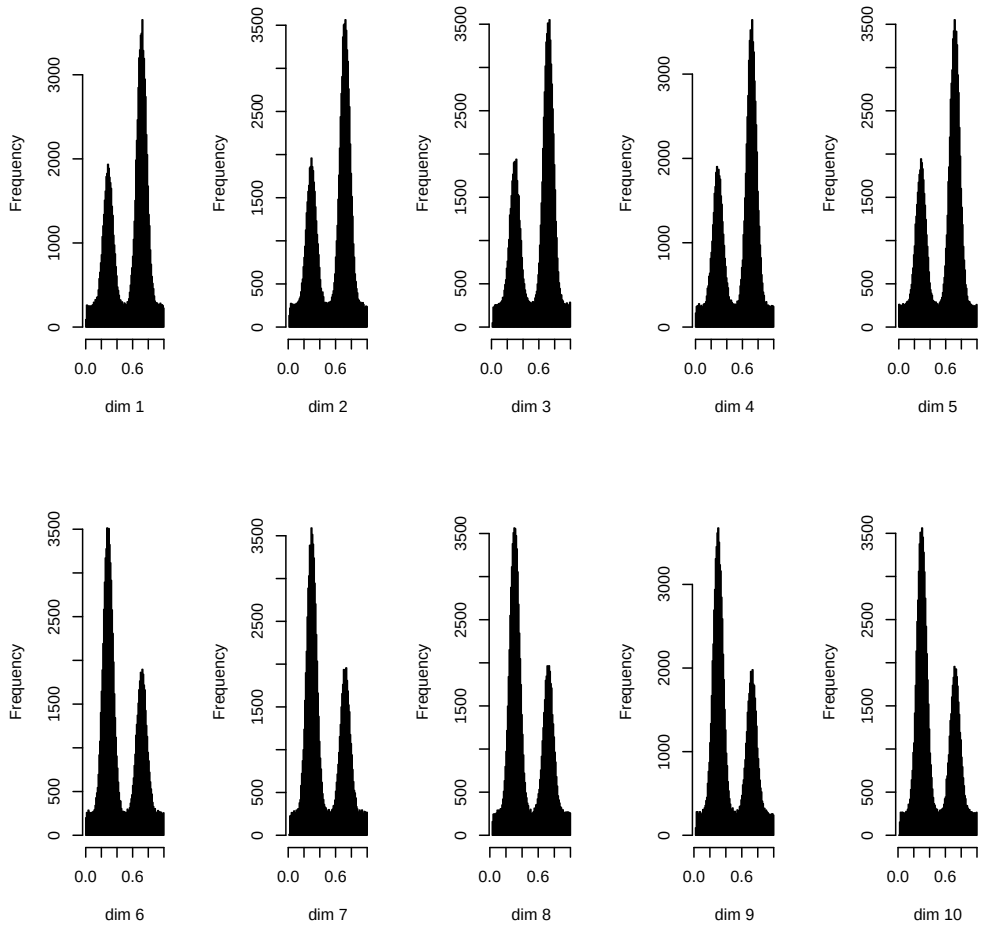


Figure 3.26: Marginal distribution of the synthetic 10 dimensional dataset

is simply due to the fact that for higher dimension group counts the replication factor does not significantly drop anymore while the number of distance computations drastically increases.

Because of the extensive replication factor, using a dimension group setting of 1 for the 15 dimensional dataset and a dimension group setting of 1 and 2 for the 20 dimensional dataset exceeded the memory capabilities of our cluster. In the first case, it aborted after about 2.7 hours and in the other two after about 2.8 and 1.2 hours. We summarize the absolute runtimes for the high dimensional datasets (10D, 15D and 20D) separately in Figure 3.27

excluding those aforementioned cases. The fastest runtime of around 4.6 minutes for the 15 dimensional dataset is achieved for a dimension group setting of four and the same setting achieves the fastest runtime for the 20 dimensional dataset of around 12.1 minutes. The θ-join approach, which does not consider dimension groups, shows higher runtimes than the PHiDJ approach for all numbers of dimension groups (7 hours for 15D and 8.75 hours for 20D).

3.7.2 Real-world data

The real dataset consists of one million 10-dimensional objects, each taking values in $[0, 1]^{10}$. They were extracted from the image database 2010-MIRFLICKR-1M. The transformation of the images into a vector space is based on pivot embedding. I.e., from the given data a small subset is chosen as pivots objects. Given an ordering of pivots, the value of dimension dim of an object o is equal to the distance of o to the dim^{th} pivot. The applied ε is 0.05.

The results for PHiDJ-static are shown in Figures 3.25(a) to 3.25(c). The qualitative results are very similar to the synthetic data. The highest runtime speedup is achieved with a dimension group number of 3 just like the 10-dimensional synthetic dataset from the last section. In this case, the program runs about 8.4 times as fast as with one dimension group. Again this behavior can be easily explained by the significant drop of the replication factor to only a 9.2-th of its original value while the number of computations only increases by a factor of 19.6. For up to 6 dimension groups, PHiDJ needs fewer computations than the θ-join approach. The runtimes of the algorithms (Figure 3.27) show a similar behavior as for the synthetic datasets.

3.7.3 Grid width

For the PHiDJ-static variant, the chosen grid width directly influences the ratio between communication costs on the one hand and the number of reducers and thus the profit from parallel distance computation on the other

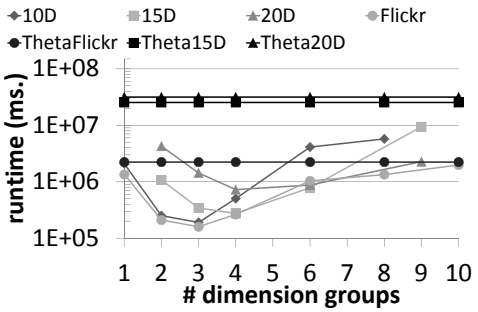


Figure 3.27: Absolute runtime (ms) on 10, 15 and 20 dimensional synthetic datasets (using PHiDJ-static)

hand. Finding a good value for the grid width can thus yield a substantial speedup. Analogous to Section 3.7.1, all datasets used in this section were created using three normal distributions with uncorrelated dimensions whose means have an Euclidean distance of at least $2 \cdot 2.5 \cdot stddev$ to each other and a simple uniform distribution for a number of noise points. The properties of the datasets are summarized in Table 3.28. The main difference between the two datasets for each dimension is the standard deviation of the normal distributions, which is 0.1 in one case and 10 in the other. $\varepsilon = 0.5$ was used for all datasets. For the 2-dimensional sets the dimension group number was set to 1 and for the 10-dimensional datasets to 4.

To test the influence of reducing the grid width on the algorithm's performance, we used the two datasets with the low standard deviation, which contain three highly compressed point clouds. With $\varepsilon = 0.5$ one has $stddev \ll \varepsilon$ and most points are concentrated in a very small fraction of all reducers. The hope is that by decreasing the grid width the profit from the increased count of reducers and thus from a higher number of parallel computations exceeds the additional communication cost since the high load from very few reducers gets distributed over a larger number of them. Unfortunately, our tests did not confirm that. While the highest speedup achieved for the 2-dimensional dataset was around 1.3 for a grid width of $0.17 \cdot \varepsilon$, the runtime dropped significantly for the 10-dimensional dataset for all reduced grid widths we tested. Apparently, the replication factor increases too fast in higher dimensions.

	2D	10D
Points:	341000	341000
Points per normal distribution:	110000	110000
Std. dev.:	0.1	0.1
Points:	1023000	1023000
Points per normal distribution:	330000	330000
Std. dev.:	10	10

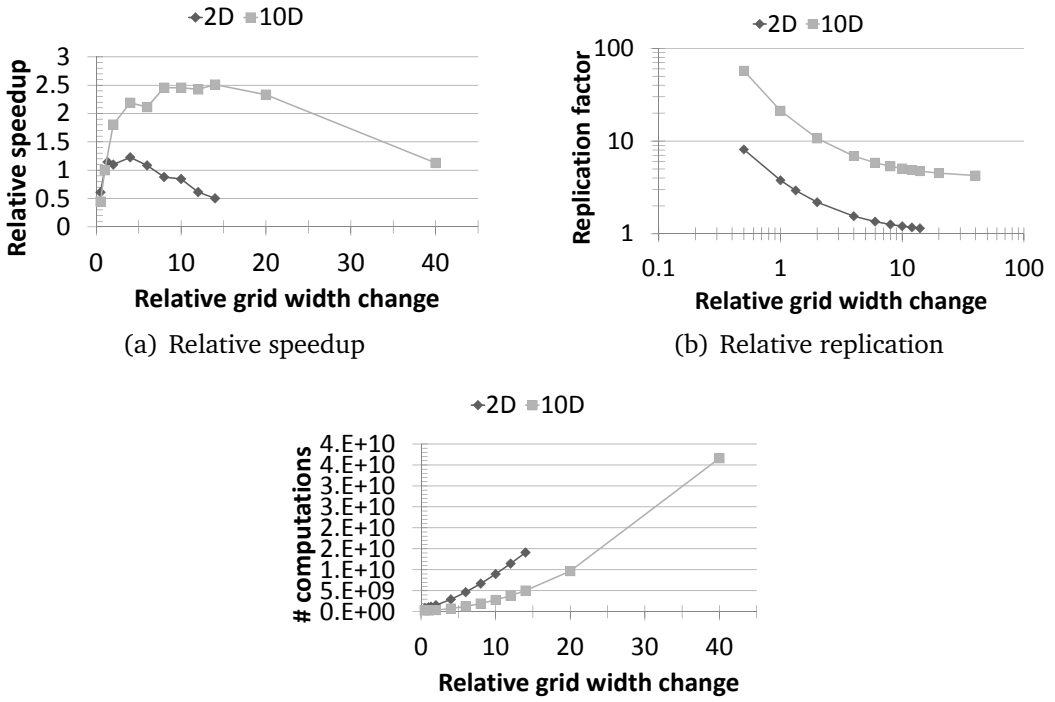
Figure 3.28: Properties of the datasets employed for the grid width evaluation

Increased grid width The two datasets with a high standard deviation have a relatively low point density, which is why each reducer gets only a relatively small number of points. By increasing the grid width the aim is that the profit from the saved communications cost exceeds the additional cost of a lower parallelization (and of a lower pruning efficiency).

Analogous to the preceding sections, Figures 3.29(a) to 3.29(c) summarize the results. The x-axis in Figure 3.29(b) (replication) now also has a logarithmic scale. In contrast to decreasing the grid width, we now have significant speedup factor of around 2.5 for the 10 dimensional dataset for a grid width factor of around 14.

3.7.4 Variable grid sizes

In this section, we investigate the influence of the variable grid sizes on the runtime, the number of calculations and the communication costs. First, we variate the number of used bins per dimension for the PHiDJ-var variant. The results of this experiment are shown in Figures 3.30(a), 3.30(b) and 3.30(c) where the number of all distance computations, the absolute replication of data and the overall runtime are depicted. Similar to non-variable grids, the increase of bin number, which is similar to changing the grid width, leads to exponential behavior in the number of computations and the data replication. The number of computations decreases, while the replication increases, for an increasing bin number. For the used ε and the given hardware, this resulted in the runtimes depicted in Figure 3.30(c). Depending on the network



(a) Relative speedup

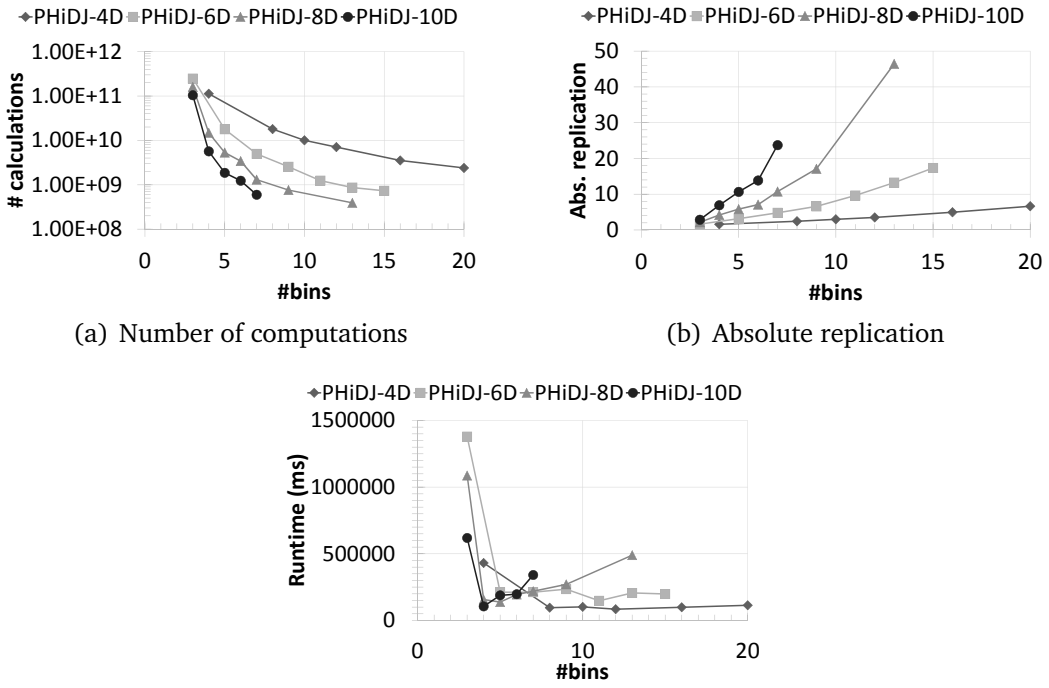
(b) Relative replication

(c) Distance computations

Figure 3.29: Experiments on low density datasets with increasing grid widths (using PHiDJ-static)

speed and the number of used CPUs this results will differ.

Next, we compare our PHiDJ-static and PHiDJ-var approaches with optimal settings against MR-SimJoin in Figures 3.31 and 3.32 on the synthetic datasets with variable dimensionality and the Flickr dataset. For the MR-SimJoin algorithm, we executed multiple experiments for determining the best settings for the given hardware and datasets. As result, we used 150 pivot points, 32 MB as split size for the MapReduce rounds and 2 MB as split size of the QuickJoin algorithm. In most cases, both PHiDJ variant clearly outperforms MR-SimJoin algorithm. The exceptions are the 6D, 20D and Flickr datasets, where MR-SimJoin either has a lower runtime than PHiDJ-static or outperforms both PHiDJ approaches. These results are also reflected in the number of calculations depicted in Figure 3.32 and the workloads of individual reducers (cf. Section 3.7.5). Please note that each dataset has an adjusted range value ε, which was chosen to keep the runtimes con-



(a) Number of computations

(b) Absolute replication

(c) Runtime

Figure 3.30: Experiment on the number of variable bins per dimension (using PHiDJ-var).

stant. This explains the fluctuations of the runtimes for different numbers of dimensions. In the last experiment, we compare the runtime of PHiDJ-var with variable grid width with the MR-SimJoin algorithm on a 20-dimensional dataset with 3, 6, 9 and 12 million points (Figure 3.34(a)). The ε value is hold constant at 0.79 and we use the optimal settings for the PHiDJ-var and the MR-SimJoin algorithm, which were determined in previous experiments. As expected, both algorithms show increasing runtimes for increasing database size. The fact that the MR-SimJoin algorithm shows a larger slope in comparison to PHiDJ-var is due to the increasing number of MapReduce jobs needed to process the data and the higher number of distance computations (Figure 3.34(b)). Apparently, the obtained optimal settings of this algorithm for the dataset with 3.1 million objects, were not optimal for larger datasets. This shows a disadvantage of the MR-SimJoin algorithm as it requires a careful parameter choice for different database sizes with even almost the same

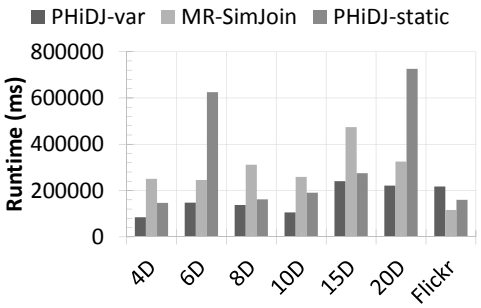


Figure 3.31: Runtime comparison between PHiDJ and MR-SimJoin

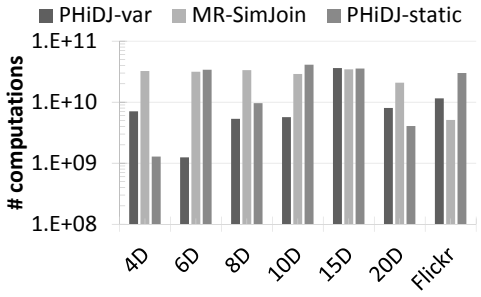


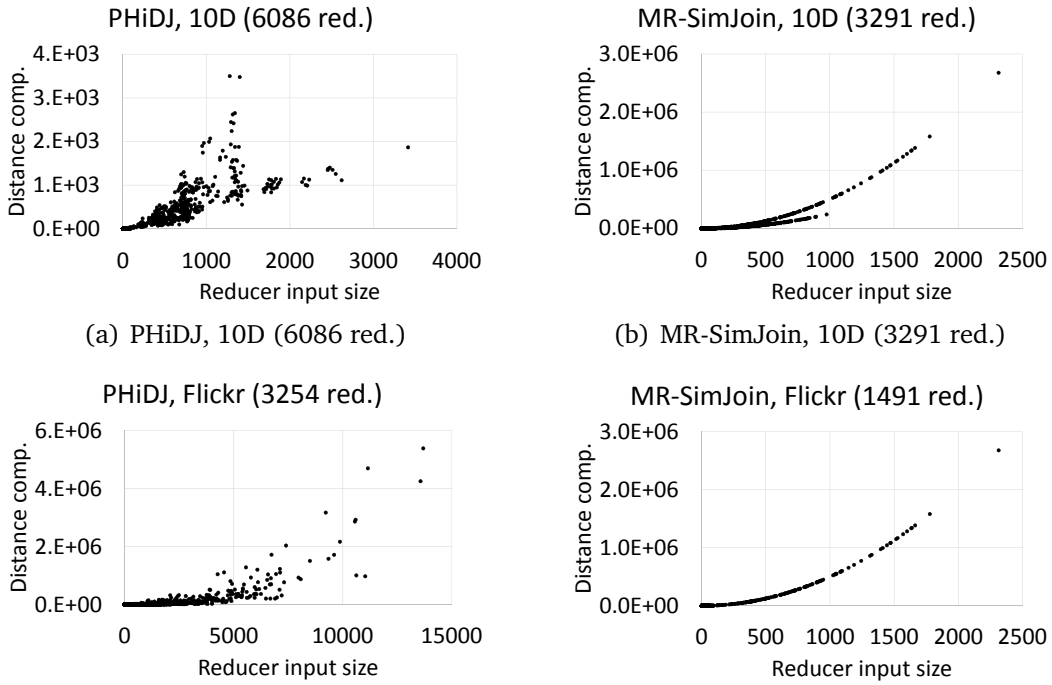
Figure 3.32: Number of computations on synthetic and Flickr datasets

data distribution.

Overall, the experimental results show that the PHiDJ algorithms and particularly PHiDJ-var should be considered as a first choice for processing similarity self-joins in medium to high-dimensional spaces.

3.7.5 Workload distribution

The last experiment addresses the workload distribution of PHiDJ-var and MR-SimJoin algorithms. For both PHiDJ and MR-SimJoin, we count the number of objects and the number of full calculations per reducer or not splittable subsets, respectively, on a 100k large subset of Flickr and 10D datasets. The results are depicted in Figure 3.33. The number of reduce tasks of PHiDJ algorithm is around two times larger as the number of unsplittable subsets in MR-SimJoin algorithm. However, for the 10D dataset, the number distance computations of reducer task with largest number of computations in PHiDJ algorithm is $\approx 10^3$ times smaller than in the MR-



(a) PHiDJ, 10D (6086 red.)

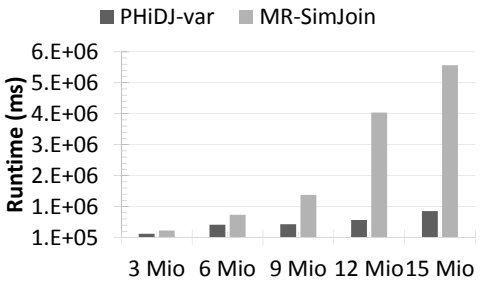
(b) MR-SimJoin, 10D (3291 red.)

(c) PHiDJ, Flickr (3254 red.)

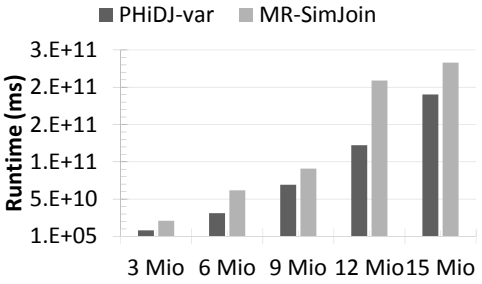
(d) MR-SimJoin, Flickr (1491 red.)

Figure 3.33: Workload distribution of PHiDJ and MR-SimJoin on 10D and Flickr datasets.

SimJoin algorithm at the same time. Among other things, this seems to be one of the main reason for worse performance of MR-SimJoin algorithm. On the other hand, for Flickr dataset, we observe the opposite behavior. The PHiDJ algorithm has multiple reduce task that need almost twice as much time as MR-SimJoin algorithm. The reason for this fact is the different data distribution of the datasets. While Flickr dataset is an almost unimodal Gaussian distribution (cf. Figure 3.14), our synthetic datasets follow a mixture of Gaussians distribution. As PHiDJ's best workload balancing is achieved for uniform distributed data, mixture of Gaussians better suits the PHiDJ algorithm design and results in better runtimes. On the other side, MR-SimJoin seems to better distribute the computations on more skewed data, as is the case for Flickr dataset.



(a) Runtime comparison between PHiDJ-var and MR-SimJoin for different database sizes.



(b) Distance computations of PHiDJ-var and MR-SimJoin algorithms for different database sizes.

Figure 3.34: Comparison of scalability of PHiDJ and MR-SimJoin algorithms.

3.8 Conclusion

In this work, we proposed the novel distance-based similarity self-join algorithm PHiDJ for the MapReduce framework. PHiDJ is well suited for medium to high-dimensional data and exploits multiple filter techniques for reducing communication and computations. We proposed a solution for processing grids with variable cell sizes, which is highly relevant for efficient join computation for skewed distributed data. The experimental evaluation for medium- to high-dimensional data shows that our approach is much more efficient than the existing grid-based solution MR-DSJ and in most cases than the MR-SimJoin algorithm.

3.9 Future work

In this section, we discuss two directions for the future work that is based on our PHiDJ algorithm. The first one relates further efficiency improvements and the second the application of the PHiDJ algorithm on other types of data.

Efficiency improvements In spite of the proposed technique for handling skewed data distributions, very dense data regions or large ε values still may badly influence the overall performance. Even reducers of very small cells have to process the data from all cells in the ε-neighborhood such that

the amount of data may remain very high. At the same time, small cells contribute to a high data replication as objects of small cells more often appear in the ε-neighborhood of other cells and vice versa.

This means, two problems must be addressed in order to further improve our solutions: workload balancing and data replication (or communication). To cope with these issues, we suggest two possible solutions. The first idea is to make the grid partitioning replication-aware, i.e., the task is to find such a partitioning that not only tries to minimize the maximal communication or number of distance computations, but also decreases the overall communication. Certainly, this solution will disbalance the workload of the join. However, as we showed in our experiments, in slow networks, the overall performance may nevertheless benefit from reduced communication.

The second approach that also addresses both problems simultaneously, is the combination of the PHiDJ algorithm with the θ-join approach[OR11] of Okcan et al.. Please remind, the θ-join computes all pairs between two relations by randomized assignment of computations to several reducers that results in a near-optimal balancing of the workload and data-independent object replication. By performing the θ-join in overfull cells of the PHiDJ algorithm in parallel way would lead to several positive effects. First, the computation of the join in very dense regions is performed by the entire cluster rather than a single reduce node. Second, the need for very small cells vanishes that on its hand reduces the data replication. Third, by exploiting the bitcode pruning, the property of the PHiDJ algorithm to compute every distance only once remains untouched. Fourth, the combination of both algorithm results in an almost optimal workload balancing.

Application on other data types Due to the underlying grid structure, the PHiDJ algorithm is strongly connected to vector data. However, many applications produce and require other data types as graphs, strings, images, and so on. Therefore, an extension of the PHiDJ algorithm for other types of data is meaningful. A possible solution for this task is the combination of the PHiDJ algorithm with *pivoting* techniques. Pivoting is a technique for transformation of objects from a metric space into the vector space. The general

procedure is simple: for each object o in the database, the distances from o to a chosen set of *pivots* are computed and each distance represents an attribute of the transformed object o'. Then, the $L_\infty(\cdot,\cdot)$-distance (maximum norm distance) between two transformed objects o', p' is a lower bound for the real metric distance between o and p. This means, having such a transformed database, the PHiDJ algorithm with the $L_\infty(\cdot,\cdot)$-distance can be applied in order to compute the superset of the real join result and then refined either in the same or in the subsequent step.

Chapter 4

CC-MR: Finding connected components in graphs

Contents

4.1 Introduction **102**

4.2 Fundamentals **103**

4.3 Related Work **104**

4.4 Algorithm **107**

4.4.1 CC-MR algorithm 108

4.4.2 Example for the processing of CC-MR 113

4.4.3 Proof of correctness 114

4.4.4 Dealing with large components 118

4.4.5 CC-MR-core: Dealing with core-/non-core objects 119

4.5 Experiments **122**

4.5.1 Scalability on synthetic data 123

4.5.2 Real-world data 125

4.5.3 Load balancing for large components 128

4.6 Conclusion **129**

In this section, we propose a novel approach for finding connected components in similarity self-join graphs with respect to core-point constraints. This is the second main step of our JBMR-DBSCAN algorithm besides the self-join phase. This step corresponds to the actual cluster determination, and as we will see later on, most of the existing parallelized approaches have a similar step that usually combines local clusters. At the same time, most of the existing approaches do not parallelize this step but perform the calculations locally on a single master node. However, when dealing with large datasets, the determination of the clusters may become a very time consuming process. In particular, this strongly holds for our JBMR-DBSCAN algorithm; in presence of large datasets, the output of the similarity self-join phase may become very large such that the parallelization of this step becomes indispensable. For that reason, in this chapter, we propose a MapReduce based solution for determination of connected components in huge undirected graphs.

Before we present our solution CC-MR-core that is able to deal with core-point constraints in Section 4.4.5, we start with the basic solution CC-MR for finding connected components in undirected graphs in Section 4.4. We shortly introduce this general topic and present further applications of this task in Section 4.1. The formal definition of the problem statement is then given in Section 4.2. Section 4.3 summarizes the existing approaches, and describes the competitor algorithms that are used for experimental evaluation in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.1 Introduction

Graph structures are a very natural way for representing complex data and therefore appear almost everywhere in the data processing; just to name a few examples: graphs are the way to represent web and social graphs, chemical compounds, protein and co-author networks, XML databases and so on. Knowledge extraction from these data often relies (at least as a preprocessing step) on the problem of finding connected components within these graphs. Due to that, besides the focus of this work on density-based clus-

tering, the horizon of applications is very broad and ranges from analysis of coherent cliques in social networks, image segmentation, where in some way connected parts of the image have to be retrieved, to database queries and many more. Thus, it is not surprising that this problem has a long research history, and different efficient algorithms were developed for its solution. Nevertheless, modern information systems produce more and more increasing datasets whose processing is not manageable on single workstations any more. Social networks like Facebook process networks with more than 750 million users* where each node is connected to 130 other nodes on average. The analysis of such enormous data volumes requires highly scalable parallelized algorithms.

In this work, we propose an algorithm for finding connected components, which is based on the MapReduce programming model and is implemented using Hadoop.

The main contributions of this chapter are:

- The parallelized algorithm CC-MR for the efficient detection of connected components in a graph using the MapReduce framework.
- Performance evaluation of the CC-MR algorithm in comparison with state-of-the-art approaches using synthetic and real-world datasets.
- Technique for improving the workload balancing of CC-MR for graphs with heterogeneous component sizes.
- The parallelized CC-MR-core algorithm for finding connected components with core point constraint

4.2 Fundamentals

In this section, we give a short formal problem definition for the general task of finding of connected components. Let $G = (V, E)$ be an undirected graph without self-loops, with V being a set of vertices and $E = \{(v, u), (u, v)\}, u, v \in$

*`http://www.facebook.com/press/info.php?statistics`, state Sep. 2011

V a set of edges. Intuitively, a connected component in G is a maximal subgraph $S = (V^S, E^S)$ in which for any two vertices $v, u \in V^S$ there exists an undirected path in G with v as start and u as end vertex. The term "maximal subgraph" means that for any additional vertex $w \in V \setminus V^S$ there is no path from any $v \in V^S$ to w.

In this work, we present a solution for finding all connected components inside the graph G. The algorithm can as well be applied to directed graphs; in this case, the result is the set of all weak connected components in the graph.

4.3 Related Work

The detection of connected components in a graph is a fundamental and well-known problem. In the past, different approaches for finding connected components were introduced. The diversity of the proposed techniques ranges from simple linear time techniques using breadth-first search or depth-first search to efficient logarithmic algorithms. Although, due to the fast growing data sizes (just think of social network graphs of Facebook or Google+), even these efficient algorithms cannot deal with such big graphs. Thus, approaches to parallelize the detection of connected components have already been developed for several decades: Hirschberg et al. [HCS79] present an algorithm that uses n^2 processors (where $n = |V|$ denotes the number of vertices in the graph) and having a time complexity of $O((\log n)^2)$. Chin et al. [CLC82] present a modified version of this algorithms which achieves the same time bound with only $n \left\lceil \frac{n}{(\log n)^2} \right\rceil$ processors. A parallel algorithm with a time bound of $O(\log n)$ is presented by Shiloach and Vishkin [SV82]. Greiner [Gre94] presents an overview of several parallel algorithms for connected components.

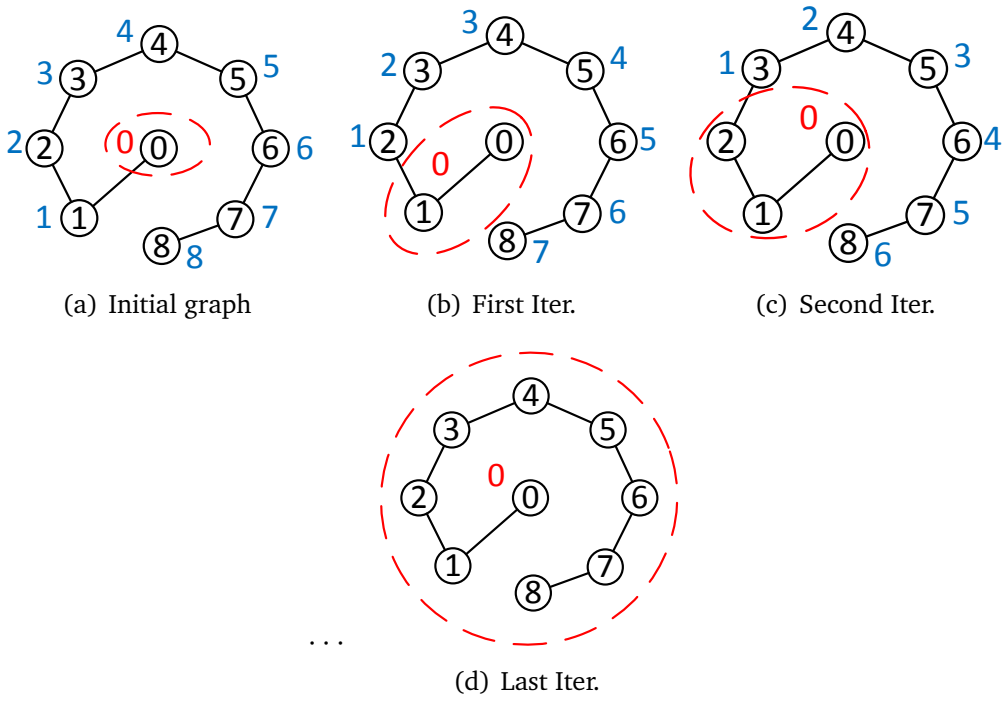
All of the aforementioned approaches assume that all computing processors have access to a shared memory and, thus, can access the same data. In contrast, the MapReduce model relies on distributing the data as well as the computation between the computing nodes and, thus, reduce the re-

quired communication between the computing nodes. Approaches for the distributed memory model are not new. For example, Bus and Tvrdik [BT01] proposed such a technique as extension of the algorithm by Krishnamurthy et al.[KLCY97]. In their distributed memory model, every computing node is able to access the memory of other computing nodes, which, however, leads to certain communication costs. In contrast, the MapReduce model only allows for special communication flows. For example, communication between different reducers in a MapReduce job is not possible. Thus, for computing connected components using MapReduce, special types of algorithms are necessary.

Recently, a few approaches for the detection of connected components using the MapReduce model were proposed. Wu et al. [WD10] present an algorithm for detecting connected components based on Label Propagation. PEGASUS [KTF09] is a graph mining system based on MapReduce and contains an algorithm for the detection of connected components. In this system, graph mining operations are represented as repeated matrix-vector multiplications. In [LMSV11] the problem is solved by finding a minimum spanning tree of the graph. For that, edges that certainly do not belong to any MST are iteratively removed until the subgraphs are small enough to be processed by a single machine. Two further algorithms were proposed in [RMCS12]. These aims at minimizing the number of iterations and communication per step. The authors provide probable bounds which are logarithmic in the largest component size but claim that in practice the number of iterations for one of the algorithms is at most $2 \log d$ (d=diameter of the graph)[†].

In [Coh09] another connected components algorithm based on MapReduce is presented. As this algorithm is the most similar one to our approach, it will be introduced in the following. In this algorithm, nodes are assigned to so-called zones, where each zone is identified by the vertex with the smallest ID contained in this zone. Initially, each node defines an own zone. The zones are then merged iteratively until finally each zone corresponds to a connected component of the graph: In each iteration, each edge is tested

[†]The extended version from the same authors was published in 2013[RMCS13] where the proof for logarithmic number of iterations was provided.



(a) Initial graph (b) First Iter. (c) Second Iter.

(d) Last Iter.

Figure 4.1: Example for the algorithm from [Coh09]

whether it connects nodes from different zones. Subsequently the algorithm finds for each zone z the zone z_{min} with the smallest ID that is connected to z and adds all vertices of z to z_{min}.

A drawback of this algorithm is that for each iteration, three MapReduce jobs have to be executed, and in each iteration, all edges of the original graph have to be processed.

In Figure 4.1, we show the processing of this algorithm for a simple example graph consisting of just one component. The numbers inside the vertices are the IDs of the vertices; the numbers beside the vertices denote the number of the zone a vertex is currently assigned to. The vertices that are already assigned to the "final" zone 0 are encircled. Initially, each vertex is assigned to its own zone. In the first iteration, the algorithm determines the edges that connect vertices from different zones, i.e., the zones 0 and 1. Then, for zone 1 the algorithm detects that the smallest zone connected to it is zone 0, i.e., the vertex 1 is now assigned to zone 0. Similarly, the vertex 2 is as-

signed to zone 1, etc. In the second iteration, the same processing is done, i.e., vertex 2 is added to zone 1, vertex 3 (former zone 2) is added to zone 1, etc. Overall, the algorithm needs eight iterations to detect the component. This example shows another drawback of the algorithm: Although in the first iteration the connection between zone 1 and zone 0 and the connection between zone 1 and zone 2 are detected, this information is not used in the second iteration. Using this information, we could for example directly add the vertex 3 from zone 2 to zone 0, as we know they are connected via zone 1 and thus have to belong to the same connected component. By neglecting this information, the algorithm needs a large number of iterations.

The basic idea of our new algorithm is to use this kind of information from previous iterations by adding additional edges ("shortcuts") in the graph such that fewer iterations are needed to find the final components. In our experimental section, we compare our approach to the approaches from [Coh09] and [KTF09].

In the meantime, Kolb et al. [KSR14] proposed an extended version of the here presented CC-MR algorithm that shows a significantly lower runtime and communication. The ideas of this work are described in more detail in Section 4.6.

4.4 Algorithm

In this section, we present our CC-MR algorithm for detecting components in large-scale graphs using the MapReduce framework. In Subsection 4.4.1, we describe our solution. For better understanding, we show in Section 4.4.2 the processing of CC-MR on the example from Section 4.3. Section 4.4.3 provides a formal proof of the correctness of CC-MR.

The basic idea of our algorithm is to iteratively alter growing local parts of the graph until each connected component is presented by a star-like subgraph, where all nodes are connected to the node having the smallest ID. For that, in each iteration, we add and delete edges such that vertices with larger IDs are assigned to the reachable vertex with smallest ID. Applying

an intelligent strategy to use the information from previous iterations, our algorithm needs significantly less iterations than existing approaches.

4.4.1 CC-MR algorithm

There are two states for a (sub)component S: either it is already maximal or there are still further subcomponents, with which S can be merged. The main question of every algorithm for finding connected components is, therefore, how to efficiently recognize those two states and how to react on them. I.e., if a component is already maximal, no further step should be performed; in the second case, the merging or some other equivalent action should be done with as little effort as possible. When dealing with parallel algorithms the question of balanced distribution of the calculations arises. Considering a distributed memory programming model like MapReduce additionally complicates the problem since an efficient information flow between independent computational nodes has to be established.

We propose a solution to handle the aforementioned states locally for every graph vertex in such a way that after at most linearly many iterations (experiments often show a logarithmic behavior for big graphs) in terms of the diameter of the largest component the solution is found. Pushing down the problem to the single vertices of the graph enables a very scalable processing in the MapReduce framework. Additionally, by using techniques for prevention of duplicated data, which often appears in distributed memory models, CC-MR algorithm significantly outperforms the state-of-the art approaches as for example [Coh09].

Let $G = (V, E)$ be an undirected graph where V is a set of vertices with IDs from $\mathbb{Z}$ and $E = \{(v_{source}, v_{dest}) \in V^2\}$ is a set of edges. The algorithm's basic idea is simple: independently check for each vertex v and its adjacent vertices $adj(v)$ whether v has the smallest ID or not. If yes (*locallyMaxState*), assign all $u \in adj(v)$ to v and stop the processing of v, since the component of v is already locally maximal. Otherwise (*mergeState*), there is a vertex $u \in adj(v)$ with $u < v$; then connect v and $adj(v)$ to u. This corresponds to assigning (merging) the component of v to the component of u. By iteratively

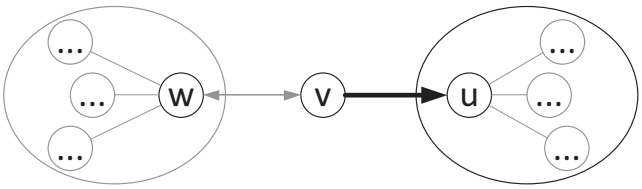


Figure 4.2: The backward edge (v, u) connects components of w and u in the reducer of v.

performing these steps, each component is finally transformed to a star-like subgraph where the vertex having the smallest ID is the center. The overall algorithm stops as far as the *mergeState* situation does not occur any more.

In the following, we present an efficient implementation based on a simple concept of *forward* and *backward edges*. We call an edge $v \rightarrow u$ a forward edge, if $v < u$, and a backward edge, if $v > u$, where the comparison of vertices means the comparison of their IDs. Both types of edges are represented by a tuple (v, u) which we represent by $(key, value)$ pairs in the MapReduce framework. The semantic of the forward edge (v, u) is that vertex u belongs to the component of the vertex v. The backward edge can be regarded as a "bridge" between the component of vertex u and the component of a vertex w which has a connection to v as shown in Figure 4.2. The backward edge between vertices v and u enables the reducer of v to connect u and w in a single iteration of the algorithm.

These concepts will become clearer from the explanation of the algorithm.

Listing 4.1: Reducer implementation.

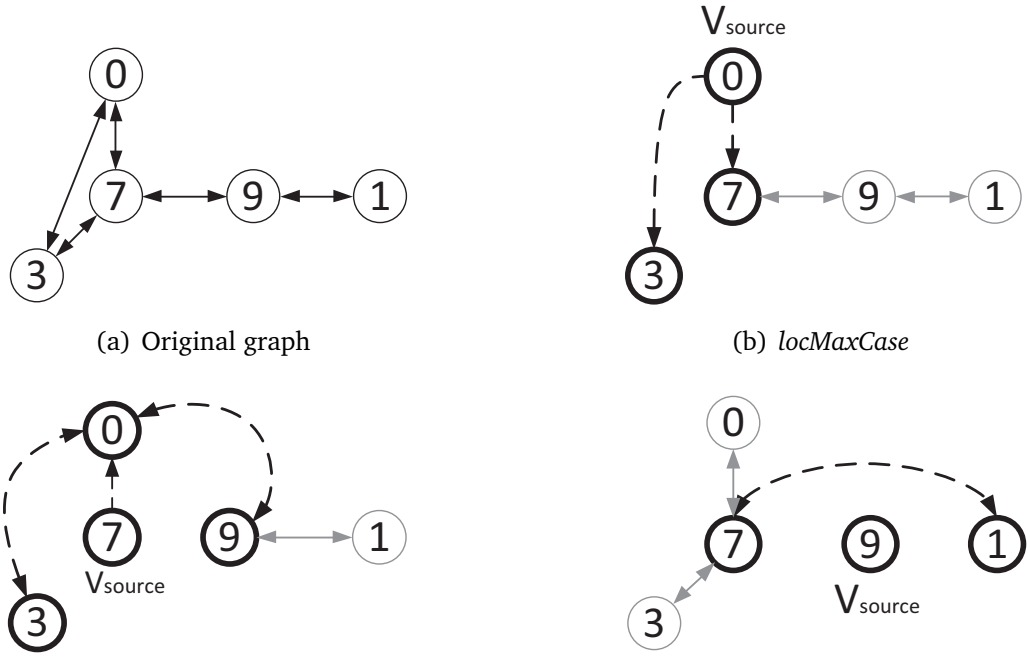
```
1 newIterationNeeded = false // global variable
2 void reduce(int v_source, Iterator<int> values)
3   isLocMaxState = false
4   v_first = values.next();   // take first element
5   if ( v_source < V_first )
6     isLocMaxState = true
7     emit(v_source, v_first)
8   v_dest_old = v_first
9   while ( values.hasNext() )
```

```
10    v_dest = values.next()
11    if ( v_dest == v_dest_old ) continue // remove duplicates
12    if ( isLocMaxState )    // locMaxCase
13      emit( v_source, v_dest ) // only fwd. edge
14    else  // cases stdMergeCase, optimizedMergeCase
15      emit( v_first, v_dest ) // fwd. edge and
16      emit( v_dest, v_first ) // backwd. edge
17      newIterationNeeded = true
18    v_dest_old = v_dest
19  // stdMergeCase
20  if ( v_source < v_dest && !isLocMaxState )
21    emit( v_source, v_first ) // backwd. edge
```

As described earlier, a MapReduce job consists of a map and a reduce phase. In our case, the mapper is a so-called identity mapper, which simply passes all read data to the reducer without performing any changes. The pseudo-code for the reduce phase is given in Listing 4.1. The emitted edges (v_{source}, v_{dest}) are automatically grouped by their v_{source} values and then sorted in ascendant order of their v_{dest}-values. Technically, we use the secondary sort method of Hadoop to establish the desired sorting. The main part of the algorithm is located in the reducer, where both aforementioned cases are handled. Tuples having the same key arrive as a data stream and are processed one after another in the 'while' loop. After the elimination of duplicate entries in line 13, three cases are distinguished:

- *locMaxCase*: lines 6–8 and 15–16
- *optimizedMergeCase*: lines 21–24
- *stdMergeCase*: lines 21–24 and 27 – 28

locMaxCase corresponds to the *locallyMaxState*, i.e., it deals with the situation when a local maximal component with root v_{source} is already found and therefore all adjacent nodes $v_{dest} \in adj(v_{source})$ have to be assigned to v_{source}. This assignment is performed by emitting forward edges $v_{source} \rightarrow v_{dest}$ in



(a) Original graph

(b) *locMaxCase*

(c) *stdMergeCase*

(d) *optimizedMergeCase*

Figure 4.3: Three cases of the algorithm. v_{source} strings marks the node under consideration in the considered reducer. Bold highlighted nodes are adjacent nodes of v_{source}.

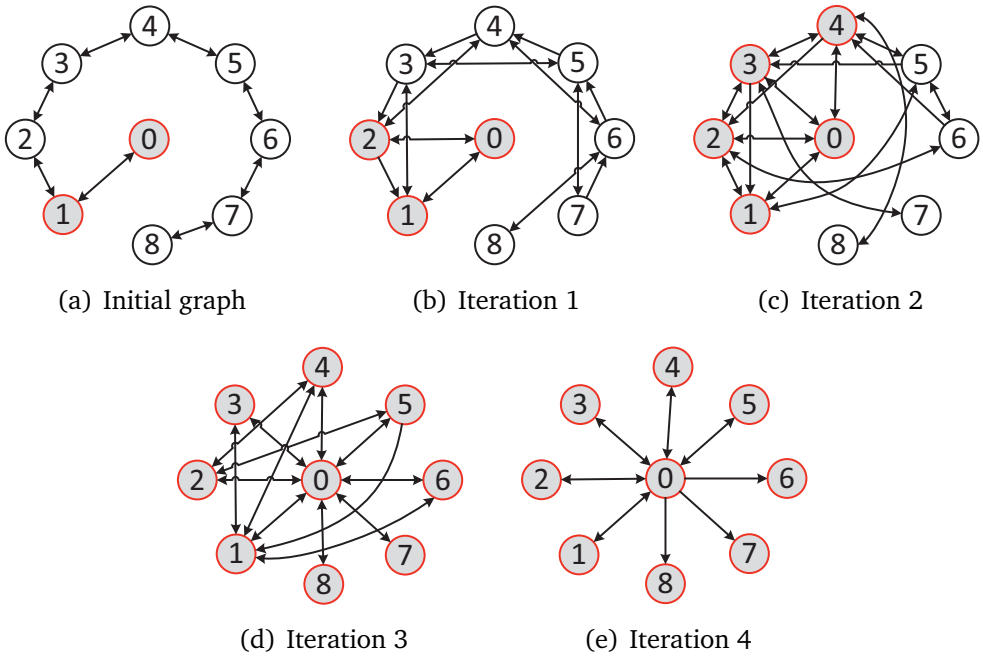
lines 8 and 16. Figure 4.3(b) depicts the processing of the case *locMaxCase* by showing the changes of the original graph structure from Figure 4.3(a). Nodes marked by v_{source} are the nodes, which are considered in single reducer with all its adjacent nodes, which for their part are highlighted by bold circles. The dimmed circles show the remaining vertices of the graph, which are not regarded during the computation of the node v_{source}. Dashed arrows represent the newly created edges inside the reducer. In this example, the reducer of the node 0 therefore emits two edges $0 \rightarrow 3$ and $0 \rightarrow 7$. Cases *stdMergeCase* and *optimizedMergeCase* on their part deal with the merge state (*mergeState*), where *optimizedMergeCase* is a special case of *stdMergeCase*, which reduces duplicate edges, as will be shown later. Both cases arise, if the condition $v_{source} > v_{first}$ holds, which means that at least one of the adjacent nodes $v_{dest} \in adj(v_{source})$ has a smaller ID than v_{source}. Because the

vertices are sorted in order of their IDs, v_{first} has the smallest value. Since the main aim of the algorithm is to assign all vertices with larger IDs to the vertex with smallest ID, this implies that all vertices in $adj(v_{source})$, except for v_{first} itself, are assigned to v_{first}; i.e., for each of these vertices a forward edge $v_{first} \rightarrow v_{dest}$ (line 21) is emitted. Please note that this is not the case for the edge $v_{first} \rightarrow v_{source}$, since this edge will be emitted in the reducer of the vertex v_{first}. Therefore, in the example of *stdMergeCase* in Figure 4.3(c) the edges $0 \rightarrow 3$ and $0 \rightarrow 9$ are emitted.

In addition to the forward edges, the algorithm emits backward edges $v_{dest} \rightarrow v_{first}$ (line 22), i.e., edges $3 \rightarrow 0$ and $9 \rightarrow 0$ in the example. These edges form "bridges" or "shortcuts" between components and are needed because v_{dest} (nodes 3, 9) could be connected to some other vertex w with even smaller ID than 0, such that at some point of time node 0 could have to be connected to w. If there were no backward edge $v_{dest} \rightarrow v_{first}$ then there would not be any reducer, which would be able to merge 0 and w.

Because of the same arguments, the backward edge $v_{source} \rightarrow v_{first}$ should be actually emitted too. This indeed happens in the case when v_{source} is smaller than one of its adjacent vertices (lines 27 and 28). If v_{source} has the biggest ID among all its adjacent nodes (*optimizedMergeCase*), then this edge can be omitted since all adjacent nodes of v_{source} are already reassigned to the vertex with smallest ID and therefore v_{source} will never deal as a bridge node between two components. In Figure 4.3(d), case *optimizedMergeCase* is depicted.

The identity mapper and the reducer from Listing 4.1 form one iteration of the CC-MR algorithm. These jobs have to be iterated as long as there are subcomponents, which can be merged. In order to recognize this case, we have to check whether in the last iteration a backward edge was emitted. If this is the case, then there are still subcomponents, which could be merged and a new iteration has to be started. Otherwise, all components are maximal and the algorithm can stop. The information whether backward edges were created or not is indicated by the global variable *newIterationNeeded*, which can be implemented as a global counter in Hadoop. Setting the value of this variable to e.g. value 1 indicates the boolean value 'true' and value 0



(a) Initial graph (b) Iteration 1 (c) Iteration 2

(d) Iteration 3 (e) Iteration 4

Figure 4.4: Example for CC-MR

indicates 'false'. This variable is set to true if either case *stdMergeCase* or case *optimizedMergeCase* holds (line 23).

4.4.2 Example for the processing of CC-MR

In Figure 4.4, we show the processing of CC-MR using the same example that was used in Section 4.3 for the algorithm from [Coh09]. For each iteration, we show the edges that the algorithm emits in this iteration. The vertices that are already connected to the vertex with the smallest ID 0 are marked in the graph for each iteration. In Table 4.1, the output of the single iterations is shown as lists of edges. For each iteration, the all edges that are emitted in this iteration are shown, sorted by their key vertices. Some edges occur repeatedly in the same iteration. This is because the same edge can be generated by different reducers.

In the initial graph, for each edge $(u, v) \in E$ both directions, i. e. (u, v) and (v, u) are given. In the first iteration, the reducers mostly insert "two-

hop" edges, e.g. the reducer for the vertex 3 connects the vertices 2 and 4. In the second iteration, for example, the reducer for the vertex 2 inserts an edge between the vertices 0 and 4 and the reducer of the vertex 6 inserts an edge between 4 and 8. Thus, vertices are already connected to each other that had a shortest path of 4 in the initial graph. In the third iteration, finally all vertices have direct connections to the node 0. However, to obtain a star graph for this component and thus to detect that the final component as already been found, the algorithm still has to delete all edges (v, w) with $v, w \neq 0$. This is done in iteration 4. Though the resulting graph is already the desired star graph, some vertices (i.e. vertex 3) still have backward edges, which are finally removed in a fifth iteration (not depicted here). Overall, our algorithm needs 5 iterations for this example. In comparison, the algorithm from [Coh09] needed eight iterations in the same example.

As one can recognize from the provided example and Table 4.1, in the first iterations, the CC-MR algorithm produces a large number of new edges in order to connect the vertices of the same component before most of the edges are removed. Similar behavior, i.e., the increase of edge number in the beginning and a fast decrease in subsequent iterations will be observed in the most real graphs. Figure 4.5 exemplary depicts the overall reducer input records for each iteration of the CC-MR algorithm on a synthetic dataset with 1000 components, each having 1500 edges and a diameter 5. As one can clearly recognize, the number of edges grows in the first iteration, and then reduces very fast, until in iterations 5 and 6 only edges of the star graphs remain.

4.4.3 Proof of correctness

In this section, we present a proof for the correctness of CC-MR, i.e. we show that CC-MR correctly detects the connected components of the graph. As presented in the previous section, the idea of our algorithm is to add and delete edges in the graph such that the resulting graph consists of one star graph per component where the center of each star graph is the vertex with the smallest ID from the corresponding component. Thus, in each iteration

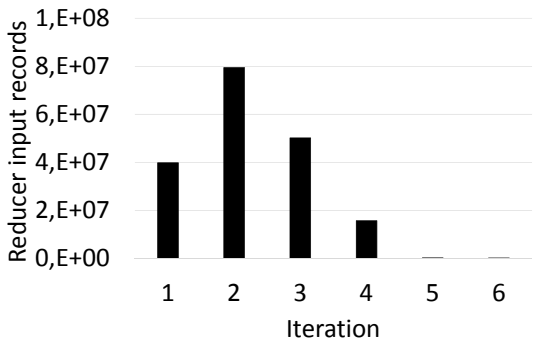


Figure 4.5: Number of edges per iteration of the CC-MR algorithm on a synthetic dataset

we have a different set of edges in the graph. Let E_i denote the set of edges that exist after iteration i.

To prove that the resulting graphs of the algorithm really correspond to the connected components, we prove two different steps:

1. An edge (v_1, v_2) is emitted in iteration $i \Rightarrow$ There has already been a path between v_1 and v_2 in iteration $i-1$.

   (We never add edges between vertices that were not in the same component before.)

2. There exists a path between v_1 and v_2 in iteration $i-1 \Rightarrow$ there exists a path between them in iteration i.

   (We do not disconnect components that existed before).

Steps 1 and 2 together show that although the algorithm adds and removes edges in the graph, the (weak) connected components do never change during the algorithm. Please note that as our input graph is undirected, the connectedness of the components does not depend on the directions of the edges, even though CC-MR sometimes only adds one direction of an edge for optimization reasons. Thus, for the paths we construct in our proof the directions of the edges are neglected. In the following, we present the proofs for the single steps:

iter.	0	1	2	3	4	5	6	7	8
0	0-1	1-0 1-2	2-1 2-3	3-2 3-4	4-3 4-5	5-4 5-6	6-5 6-7	7-6 7-8	8-7
1	0-1 0-2	1-0 1-3	2-0 2-1 2-4	3-1 3-2 3-5	4-2 4-3 4-6	5-3 5-4 5-7	6-4 6-5 6-8	7-5 7-6	8-6
2	0-1 (2x) 0-2 0-3 0-4	1-0 1-0 1-2 1-5	2-0 2-1 2-3 2-6	3-0 3-1 3-2 3-4 3-7	4-0 4-2 4-3 4-5 4-8	5-1 5-3 5-4 5-6	6-2 6-4 6-5	7-3	8-4
3	0-1 (3x) 0-2 (4x) 0-3 (3x) 0-4 (2x) 0-5 (2x) 0-6 0-7 0-8	1-0 1-0 1-0 1-3 1-4 1-6	2-0 2-0 2-0 2-0 2-4 2-5	3-0 3-0 3-0 3-1	4-0 4-0 4-1 4-2	5-0 5-0 5-1 5-2	6-0 6-1	7-0	8-0
4	0-1 (5x) 0-2 (3x) 0-3 (2x) 0-4 (3x) 0-5 (2x) 0-6 (2x) 0-7 0-8	1-0 1-0 1-0 1-0 1-0	2-0 2-0 2-0	3-0	4-0 4-0	5-0	6-0		
5	0-1 0-2 0-3 0-4 0-5 0-6 0-7 0-8								

Table 4.1: Edges generated by CC-MR for the example graph

1. In CC-MR, edges are only added in the reducers. Thus, to add an edge $(v_1, v_2) \in E_i$, the vertices v_1 and v_2 have to occur together in the reducer of some vertex v_{source}. Therefore, for each $v_j, j \in \{1, 2\} : v_j = v_{source}$ or $(v_{source}, v_j) \in E_{i-1}$. Thus, there existed a path between v_1 and v_2 in the iteration $i - 1$.

2. It suffices to show: There exists an *edge* $(v_1, v_2) \in E_{i-1} \Rightarrow$ there exists

a path between them in iteration i (Because a path between some vertices u and w in E_{i-1} can be reconstructed in E_i by replacing each edge $(v_1, v_2) \in E_{i-1}$ on the path by its corresponding path in E_i):

Case 1: $v_1 < v_2$ (i.e. (v_1, v_2) is a forward edge):

(v_1, v_2) is processed in the reducer of $v_{source} = v_1$. Now we can look at the three different cases that can occur in the reducer:

- *locMaxCase*: (v_1, v_2) is emitted again.
- *stdMergeCase*: We emit edges (v_{first}, v_2), (v_2, v_{first}) and (v_1, v_{first}), thus there still exists a path between v_1 and v_2.
- *optimizedMergeCase*: Not possible because $v_1 < v_2$.

Case 2: $v_1 > v_2$ (i.e. (v_1, v_2) is a backward edge):

For this case, we can show that in some iteration $i_x \leq i - 1$ also the corresponding forward edge (v_2, v_1) was emitted:

For the backward edge $(v_1, v_2) \in E_{i-1}$, there are two possible scenarios where (v_1, v_2) can have been emitted:

- $(v_1, v_2) \in E_{i-1}$ can have been emitted by the reducer of v_1 in the case *stdMergeCase*. In this case, the edge has already existed in the previous iteration , i.e. $(v_1, v_2) \in E_{i-2}$, else the vertex v_2 would not be processed in the reducer of v_1.
- $(v_1, v_2) \in E_{i-1}$ can have been emitted by the reducer of a vertex v_{source} with $v_{source} \neq v_1$ and $v_{source} \neq v_2$ in the case *stdMergeCase* or *optimizedMergeCase*. In this case, also the forward edge (v_2, v_1) has been emitted.
- $i - 1 = 0$, i.e. the edge (v_1, v_2) already existed in the initial graph. Then, by definition of the original (undirected) graph, also the forward edge (v_2, v_1) existed.

Thus, we know that for each backward edge (v_1, v_2), the corresponding forward edge (v_2, v_1) exists or has existed in an earlier iteration. In case

1 it was already shown that the path between v_2 and v_1 is preserved in the following iterations. ■

4.4.4 Dealing with large components

In CC-MR, a single reducer processes a complete component, which can result in high workloads of a single reducer for very large components. Now we briefly present a solution, which distributes the calculations for large components over multiple reducers and that way balances the workload. Consider the example in Figure 4.6, in which vertex 7 is a center of a component with too many elements. Assume that we have this information from a previous iteration and we want to distribute the computations on three reducers. For

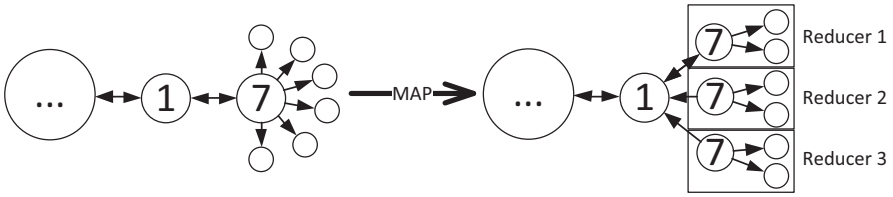


Figure 4.6: Example for workload balancing for big components.

that, in the map-phase each forward edge $(7, x)$ is augmented by a hash value to $((7, hash(x)), x)$ which is then used in the partitioner in order to distribute edges $(7, \cdot)$ to different reducers. In the example, a small circle represents such a vertex x, which is then sent to one of the three reducers. For backward edges (in the example $(7, 1)$), a set of edges $\{((7, hash(i)), 1) | i = 1, 2, 3\}$ is produced, which guarantees that each vertex 7 in each of the three reducers has a backward edge to 1 and can reassign its neighbors to 1 in the reduce phase. All other edges whose source vertices do not have too many neighbors are not augmented by a hash value and therefore are processed as in the original algorithm.

The reduce-phase remains as in the original algorithm, with the difference that for each vertex, the number of neighbors in E_i is determined and for vertices with too many neighbors the value is stored in the distributed cache for

the next iteration. This simple strategy produces almost no additional overhead but achieves a very good balancing of the workload as will be shown in the experiments.

4.4.5 CC-MR-core: Dealing with core-/non-core objects

Until now, the presented solution was not able to differentiate between core and non-core objects as it is required by the DBSCAN algorithm. In this section, we propose a simple extension for the CC-MR algorithm - the *CC-MR-core* - that considers this information such that given a similarity graph $G_{\bowtie_\varepsilon}$ and information whether a point is a core or a non-core point, the connected components correspond to desired clusters of the DBSCAN algorithm.

In the following, we assume that the information whether a point is a core point or not is already given and is encoded by the sign of the vertex ID. A negative value of a vertex v means that v is a non-core object and positive value means that v is a core point. Moreover, there are no edges $(-v, u)$, i.e., no edges where a non-core object v has a forward edge to some other vertex u. The MapReduce job for determining and encoding the core/non-core objects in such a way is presented in Section 5.2. We call the graph $G^{core}_{\bowtie_\varepsilon}$ that has such a structure a *core similarity graph*.

Definition 4.1 (Core similarity graph $G^{core}_{\bowtie_\varepsilon}$) *Let $G_{\bowtie_\varepsilon} = (V, E)$ be a similarity graph (cf. Definition 2.10), $Core = \{v \in V || NN_\varepsilon(v)| \geq minPts\}$ be the set of core objects in V and $NonCore = \{v \in V || NN_\varepsilon(v)| < minPts\}$ be the set of non-core objects in V. The corresponding* core similarity graph $G^{core}_{\bowtie_\varepsilon} = (V', E')$ *is defined as:*

- $V' := \{v | v \in Core\} \cup \{-v | v \in NonCore\}$
- $E' := \{(u, v) | (u, v) \in E \wedge u \in Core \wedge v \in Core\} \cup \{(u, -v) | (u, v) \in E \wedge u \in Core \wedge v \in NonCore\}$

Given the core similarity graph, the CC-MR algorithm can be easily extended to handle core and non-core points. Call in mind that the DBSCAN algorithm only then expands an existing cluster $C \subseteq DB$, i.e., assigns further objects

to C, if at least one of the points $p \in C$ is a core object and p is connected to some point $o \in DB \backslash C$ not belonging to C. This corresponds to the statement that two clusters $C_1 = \{o|o \in DB\} \subseteq DB$ and $C_2 = \{q|q \in DB\} \subseteq DB$ are combined if and only if at least one core point $o \in C_1$ is a neighbor of a core point $q \in C_2$, i.e., $o \in NN_\varepsilon(q)$ (or vice versa), i.e., if there is a "bridge" between C_1 and C_2.

The structure of the CC-MR algorithm allows a simple extension to support the $minPts$-restriction for merging objects. By construction of the core similarity graph, no non-core object is a source of any edge $(src, dest)$, therefore, none of the three $locMaxCase$, $stdMergeCase$ or $optimizedMergeCase$ case must consider this situation.

By sorting the values received by a reducer in an ascending order and at the same time sorting the negative values in the end, the $locMaxCase$ and $optimizedMergeCase$ remains unchanged. Only the $stdMergeCase$ must be extended by an additional check, whether a currently processed vertex is a core point or not. In the first case, the standard implementation is executed, i.e., the forward and backward edges are emitted. In the latter case, only a forward edge from the vertex with smallest ID to the non-core vertex is emitted. The described extension is shown in Listing 4.2 of the reduce function and in Theorem 4.1 the proof for the correctness of the algorithm is given. The gray lines in the code highlight the changes of the code. While in the if-statement in line 6 only the second condition is added in order to deal with the case when all values are non-core objects, lines 18–20 are new and add the above-described change of the $stdMergeCase$ case.

Listing 4.2: Reducer for CC-MR-core algorithm.

```
1 newIterationNeeded = false // global variable
2 void reduce(int v_source, Iterator<int> values)
3   isLocMaxState = false
4   v_first = values.next();   // take first element
5
6   if ( v_source < V_first || V_first < 0 ) // recognize the locMaxState
7     isLocMaxState = true
8     emit(v_source, v_first)
```

```
9
10  v_dest_old = v_first
11  while ( values.hasNext() )
12    v_dest = values.next()
13    if ( v_dest == v_dest_old ) continue // remove duplicates
14
15    if ( isLocMaxState )    // locMaxCase
16      emit( v_source, v_dest ) // only forward. edge
17    else  // cases stdMergeCase, optimizedMergeCase
18      if ( v_dest < 0 ) // non−core point
19        emit( v_first, v_dest )
20      else // core points
21        emit( v_first, v_dest ) // forward edge and, also to non−core objects
22        emit( v_dest, v_first ) // backward edge
23        newIterationNeeded = true
24    v_dest_old = v_dest
25
26  // stdMergeCase
27   if ( v_source < v_dest && !isLocMaxState )
28    emit( v_source, v_first ) // backward edge
```

Please note that following the presented idea, clusters with not disjunct point sets may occur if a non-core point p is connected to multiple clusters. In this case, all clusters will contain p in the result set. The original DBSCAN algorithm solves this multiple-assignment-problem by assigning p to the first cluster that is processed by the algorithm. However, this also means that p will be assigned to a different cluster if another ordering of the objects is used. As we do not consider this case to be problematic, we do not deal with this case any more.

Theorem 4.1 (Correctness of the CC-MR-core algorithm.) *For a given core similarity graph $G^{core}_{\bowtie_\varepsilon}$ constructed for a user-specified parameter $minPts = k$, the output of the CC-MR-core algorithm corresponds to the DBSCAN clustering with $minPts = k$.*

Proof 4.1 *The proof of the theorem bases on four facts:*

1. *The CC-MR-core algorithm processes the core objects in the same way as the CC-MR algorithm. As a connected component of core objects corresponds to a density reachable set of core points, the computed connected components of core points corresponds to the desired DBSCAN clusters.*

2. *The CC-MR-core algorithm does not emit any backward edges with a non-core point object as a key, i.e., no $(-v, u)$ edges (lines 18–20). Therefore, no non-core vertex can combine two sub-components that corresponds to the statement, that no non-core object merges two clusters.*

3. *A core object u emits a forward edge to a non-core object v, if there is an edge $(u, -v)$ (line 21), i.e., the border object v will be assigned to some cluster and the density reachability condition of a DBSCAN cluster is preserved.*

4. *Finally, by construction of the $G^{core}_{\bowtie_\varepsilon}$ graph, none of the non-core objects is a source of an edge, such that none of the reducers processes the non-core objects. Therefore, if a set of non-core objects are connected among each other or not connected at all, they are removed from processing after the first iteration of the CC-MR-core algorithm. As the CC-MR-core algorithm performs at least a single iteration, all noise objects will be removed. However, they can be easily recomputed by computing the disjunction of original dataset and the clustering result.*

Summarized, the points 1, 2, 3 the CC-MR-core algorithm guarantee the computation of correct DBSCAN clustering and the point 4 shows that the noise objects do not affect the result. ■

4.5 Experiments

In this section, we present the experimental evaluation of the CC-MR approach, comparing it to the approach from [Coh09] (denoted as 'GT' here) and to the approach from the Pegasus system [KTF09]. All experiments were

conducted on a cluster running Hadoop 0.20.2 and consisting of 14 nodes with 8 cores each that are connected via a 1 Gbit/sec. network. Each of the nodes has 16 Gb RAM. For each experiment, the number of distance computations and the runtime is measured[‡].

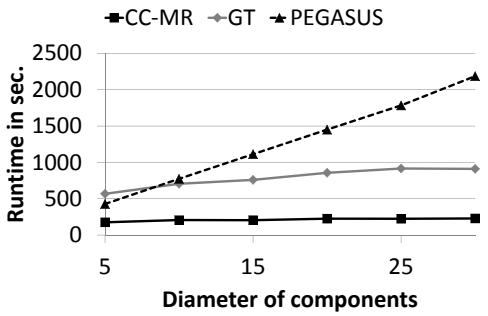
4.5.1 Scalability on synthetic data

In this section, we evaluate CC-MR with respect to different properties of the input graphs. Therefore, we use synthetic datasets such that in each experiment, one property of the generated graphs changes while the others remain stable.

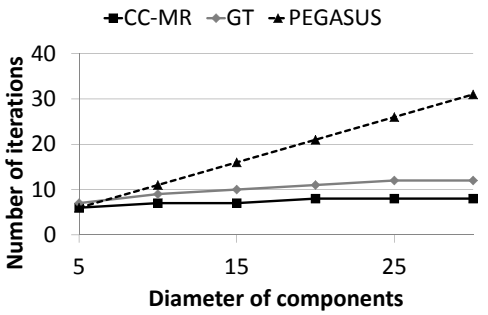
Performance for varying component diameters In this experiment, we vary the diameter of the connected components in our synthetic datasets. Each dataset consists of one million vertices and is divided into 1000 connected components with 1000 vertices each. The diameter of the generated components is varied from 10 to 30 in this experiment.
In Figure 4.7(a), the runtime of the algorithms is depicted. For all algorithms, the runtime increases for higher diameters. However, the runtime of CC-MR is always significantly lower (at least a factor of two) than that of GT and Pegasus and scales better for increasing diameters. In Figure 4.7(b), we show the number of iterations needed by the algorithms to find the connected components. As expected, for higher diameters the number of iterations increases for all algorithms. For CC-MR, the number of iterations is always lower than for GT and Pegasus. In Figure 4.7(c), we depict for each algorithm the number of input records (summed over all iterations) that are processed, i.e., the number of records that are communicated between the iterations. For all algorithms, this number increases with increasing diameter. The number of records for CC-MR is significantly lower (at least by a factor of two) than that of GT and Pegasus, because they perform more iterations than CC-MR, and GT needs to perform 3 MapReduce jobs per iteration.

Performance for varying component sizes In this experiment, we vary

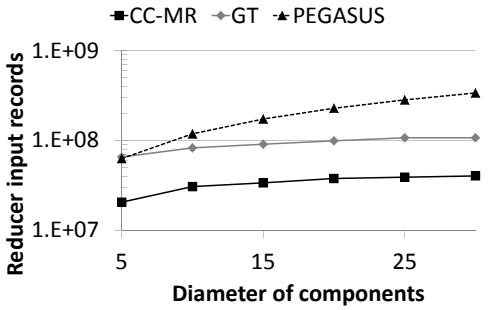
[‡]The source code of the CC-MR algorithm and the used datasets can be found at: `http://dme.rwth-aachen.de/en/ccmr`



(a) Component diameter vs. runtime



(b) Component diameter vs. number of iterations

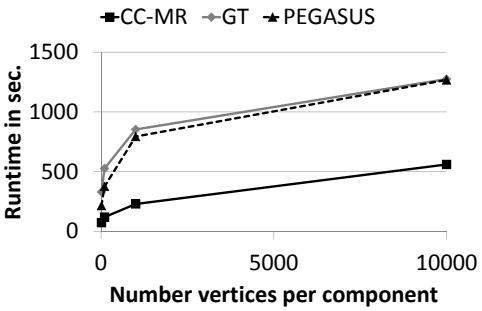


(c) Component diameter vs. number of input records

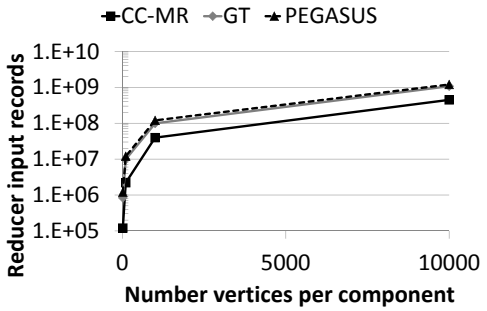
Figure 4.7: Performance for varying component diameters

the number of vertices per component in the generated graphs and examine its influence on the runtime and the number generated records in reducers. The created datasets consist of 1000 components with 15, 150, 1500 and 15000 edges and 10, 100, 1000, 10000 vertices, respectively. Figures 4.8(a), 4.8(b) and 4.8(c) depict the results for the runtime (in sec.), the number of records, and the iteration number respectively. As expected, all of these values increase with growing size of the components. The runtime of the CC-MR algorithm remains smaller (up to a factor of 2) then the runtimes of GT and Pegasus for each component size.

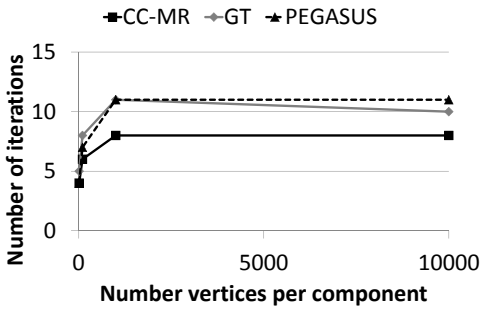
Performance for varying numbers of components This experiment shows the dependency between the number of components in a graph, the runtime, the number of input records and the number of iterations. The synthetic datasets consist of 10, 100, 1000, 10000 components each with 1000 vertices



(a) Component size vs. runtime



(b) Component size vs. number of input records



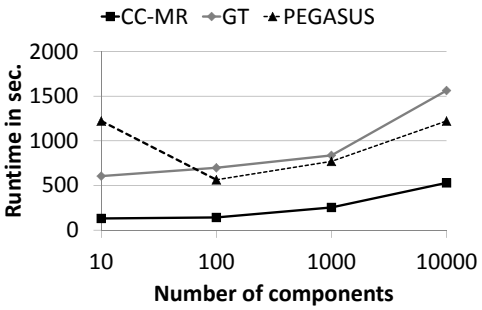
(c) Component size vs. number of iterations

Figure 4.8: Performance for varying component sizes

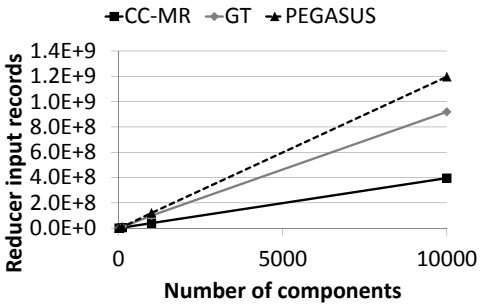
and 1500 edges. The Figures 4.9(a), 4.9(b) and 4.9(c) depict the runtime, the processed number of input records and the number of iterations. Similar to previous results, CC-MR has a much lower runtime, and produces in the worst case (10 components) 15% and in the best case (10000 components) over 55% less input records compared to the other approaches. Furthermore, CC-MR outperforms both competitors in terms of the number of performed iterations.

4.5.2 Real-world data

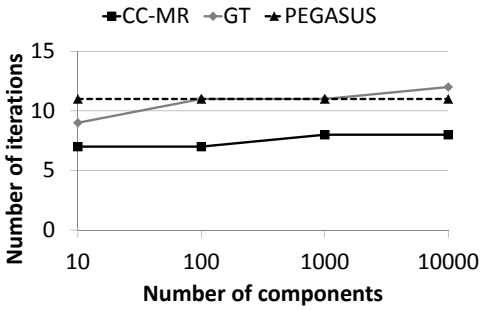
We use three real-world datasets to evaluate CC-MR: a web graph (Web-google) and two collaboration networks (IMDB, DBLP). The Web-google dataset can be found at `snap.stanford.edu/data/web-Google.html` and consists of 875713 nodes, 5105039 edges and 2746 connected components.



(a) Number of components vs. runtime



(b) Number of components vs. number of input records



(c) Number of components vs. number of iterations

Figure 4.9: Performance for varying numbers of components

In this web graph, the vertices represent web pages and the edges represent hyperlinks between them. In the IMDB dataset (176540 nodes; 19992184 edges; 16 comps.), actors are represented by vertices, edges between actors indicate that the actors worked together in some movie. The data was extracted from the IMDB movie database (`imdb.com`). In the DBLP dataset (generated using the data from `dblp.org`, 553797 nodes; 1677487 edges; 24725 comps.) each vertex corresponds to an author, while each edge represents a collaboration between two authors.

The results of the comparison are given in Table 4.2. CC-MR clearly outperforms GT and PEGASUS in terms of the number of iterations as well as in the runtime and the communication (i.e the number of input records).

Figure 4.10 depicts the scalability results of the evaluated algorithms. The CC-MR-algorithm shows a speedup more than twice as high compared with competing approaches. As Figure 4.11 shows, the reduce time of our

	CC-MR		
	web-google	imdb	dblp
Runtime (sec.)	535	1055	472
#iters	8	6	7
#input rec. ($\cdot 10^6$)	29	179	27
	GT		
	web-google	imdb	dblp
Runtime (sec.)	3567	3033	4385
#iters	10	6	12
#input rec. ($\cdot 10^6$)	102	564	210
	PEGASUS		
	web-google	imdb	dblp
Runtime (sec.)	4847	2834	3693
#iters	16	6	15
#input rec. ($\cdot 10^6$)	292	299	108

Table 4.2: Results of CC-MR, GT and PEGASUS.

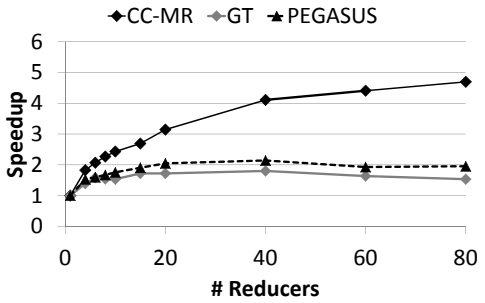


Figure 4.10: Speedup for varying number of reduce nodes

approach decreases very fast with growing number of reducers. The moderate overall speedup of 4.7 with 80 reducers puts down to the fact that the map phase does not depend on the number of used reducers and therefore the speedup of calculations is limited by I/O speed. The significant communication reduction of our approach is therefore a very big advantage in comparison to the competing approaches.

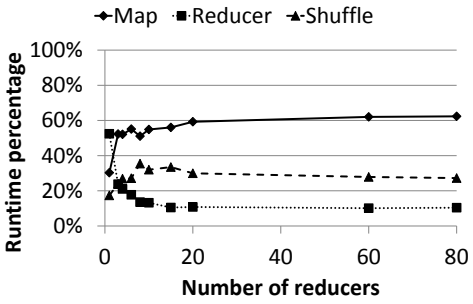


Figure 4.11: CC-MR-Runtime distribution among map, shuffle and reduce phases.

4.5.3 Load balancing for large components

Figure 4.12 shows the load balancing properties of our large component extension (cf. Section 4.4.4) on the IMDB dataset using 20 reducers and threshold for the maximal size of a component set to 1% of the number of edges in an iteration. This dataset contains 16 components, 15 small ones and one, C_{max}, containing more than 99% of the vertices. In the first three iterations both algorithms perform similarly well and distribute the workload almost evenly among all reducers. In iteration 4, however, the reducer R_{max} of the CC-MR responsible for C_{max} already processes about 50% of all edges remaining in the iteration, as more and more vertices are assigned to C_{max}. This trend goes on until in the last iteration almost all vertices of the graph are processed by R_{max}. In contrast, our extended algorithm is able to balance the workload after each iteration when a disbalance occurs, as it is the case in the iterations 3 and 5.

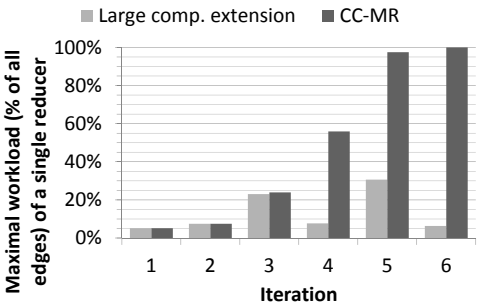


Figure 4.12: Maximal reducer workload for processing large components.

4.6 Conclusion

In this thesis, we propose the parallel algorithm CC-MR for the detection of the connected components of a graph. The algorithm is built on top of the MapReduce programming model. CC-MR effectively manipulates the graph structure to reduce the number of needed iterations and thus to find the connected components more quickly. Furthermore, we propose an extension to CC-MR to deal with heterogeneous component sizes. Apart from the description of the algorithm, we also provide a proof for the correctness of CC-MR. The performance of CC-MR is evaluated on synthetic and real-world data in the experimental section. The experiments show that CC-MR constantly outperforms the state-of-the-art approaches.

As was stated in Section 4.3, in the meantime, Kolb et al. [KSR14] proposed an extension of the CC-MR algorithm. We shortly introduce some of the main improvements of this work. As the performance of the CC-MR algorithm highly depends on the number of iterations and the intermediate communication, the main goal is to reduce these values.

First proposed improvement exploits the fact that a center vertex v with high degree of some component C will usually faster accumulate the vertices of C than a center vertex with a low degree, since more vertices of C will be assigned to v in the first step. This idea is implemented in a heuristic CC-MR-VD, which uses a light-weight preprocessing MR-job to determine the vertex degrees and then exploits the distributed cache mechanism of MapReduce framework to distribute this information to all nodes. Finally, the vertices are annotated by the determined degrees and such that in the shuffle phase both value - the degree and vertex id - are used for appropriate sorting.

Second idea introduces a local component determination in the map phase, i.e., the mapper store a user-specified number of edges of the graph and find the local components in these partial subgraph. Then, instead of emitting the whole subgraph, only the reduced subresults are emitted to the reducers. This results in lower overall communication, smaller size of the data in the shuffle phase and lower workload of the reduce nodes.

Finally, Kolb et al. propose to remove components from further process-

ing if they are already complete. This case appears in graphs with components of different sizes. While small components are determined in a few iterations, large components require more steps in order to be identified.

The experimental evaluation of the proposed changes shows a significant runtime improvement and communication reduction on the used datasets.

Chapter 5

DBSCAN in MapReduce

Contents

5.1 Related work **132**
5.1.1 Parallel non-MapReduce-based approaches . . . 132
5.1.2 MapReduce-based approaches 134
5.2 JBMR-DBSCAN approach **136**
5.3 Integrated JBMR-DBSCAN approach **139**
5.3.1 Basic procedure 140
5.3.2 Implementation in MapReduce 145
5.3.3 Proof of correctness 148
5.4 Evaluation **150**
5.4.1 Evaluating JBMR-DBSCAN 152
5.4.2 Evaluating JBMR-DBSCAN and competitive approaches 154
5.5 Summary and conclusion **157**
5.6 Future work **158**

In Chapter 2, we described the DBSCAN clustering model and presented the basic idea of our novel solution. However, the concrete MapReduce implementation was left without attention. For that reason, in this chapter, we

finally introduce our novel MapReduce-based approach JBMR-DBSCAN for the DBSCAN algorithm.

The chapter begins with a description of existing MapReduce-based and parallel non-MapReduce-based approaches in Section 5.1. Then, in Section 5.2, our novel MapReduce-based solution JBMR-DBSCAN is presented. An integrated version of JBMR-DBSCAN that significantly decrease the runtime of the algorithm is proposed in Section 5.3. The efficiency of both approaches is compared with previously introduced competitive techniques in Section 5.4. Finally, Section 5.5 concludes this chapter, and in Section 5.6 possible extensions for the future research are proposed.

5.1 Related work

The parallelization of the DBSCAN algorithm gained a lot of attention in the scientific area. In this section, we list and describe several solutions covering non-MapReduce-based techniques in Section 5.1.1 as well as MapReduce-based solutions in Section 5.1.2. The original implementation as proposed by Kriegel et al.[EKSX96] was already described in Section 2.1. Therefore, we immediately dive into the parallelization approaches.

5.1.1 Parallel non-MapReduce-based approaches

In this section, we describe the non-MapReduce-based solutions for the parallelization of the DBSCAN algorithm. Xu et al. proposed one of the first approaches PDBSCAN ([XJK99]) for a network-connected shared-nothing architecture in 1999. Its main idea is to partition the dataset using the leaves of an R*-tree into almost equal subsets $S_1, \ldots, S_n, \bigcup S_i = DB$, which contain spatially nearby objects and to process the single subsets in parallel. After determination of the local results, the subresults of the slave nodes are connected in a single master node. To accelerate the range query processing and to reduce the communication of the algorithm, the authors introduce and employ the dR+-tree index structure. The dR*-tree is a distributed variant of the well-known R*-tree index structure [BKSS90], whose directory is

replicated among all computational nodes and the data pages are distributed on the available computers.

Zhou et al. [ZZC+00] proposed an approach for shared-memory multiprocessor systems in 2000. As PDBSCAN, the proposed algorithm follows the data parallelism paradigm, i.e., the data is distributed across multiple computing nodes that perform the DBSCAN clustering algorithm. The subresults of the single nodes are then gathered and combined by a central master node. To evenly distribute the data, the authors propose the usage of a grid partitioning. In contrast to the PDBSCAN algorithm, the R*-tree index for accelerating the query processing is built for every cell. To further accelerate the single parallel tasks, the authors propose a sample-based approximate DBSCAN algorithm. However, all introduced techniques are evaluated and presented for two-dimensional datasets only. As the authors state, the extension of the algorithm for high-dimensional spaces is possible, but will result in a significantly more complex underlying algorithm.

Arlia and Coppola [AC01] show how the DBSCAN algorithm can be parallelized in the programming environment SkIE. However, they do not introduce any novel concepts but rely on a master-slave model, where the slaves perform the query processing and the master node labels the retrieved objects.

Januzaj et al. [JKP04] propose the DBDC - Density Based Distributed Clustering algorithm that resembles the DBSCAN algorithms in many ways. Similar to PDBSCAN, the algorithm follows the master-slave parallelization paradigm where the slave nodes compute local clustering results of data subsets and the master node combines the partial results. However, in contrast to PDBSCAN or similar algorithms, the authors assume the data being already distributed as it is a usual case for distributed file systems or distributed databases. The focus of that work is a precise but memory-efficient representation of local clusterings and an effective merging strategy of such representations to a global clustering. The result of the DBDC algorithm does not exactly equal the DBSCAN clustering as the merging procedure relies on aggregated clustering representations only, such that DBDC can be considered as an approximate solution of the original DBSCAN algorithm.

In contrast to previous approaches, the P-DBSCAN algorithm [CGL10], presented by Chen et al. in 2010, employs a PR-tree [AdBHY04] instead of an R*-tree for efficient query processing in its computational nodes. However, as no further novel ideas were introduced, we do not consider this approach in any further details.

Mostofa et al. [PPA+12] proposed an algorithm called PDSDBSCAN that employs graph-algorithmic concepts for the parallelization of the DBSCAN algorithm in the master-slave and shared-memory architectures. The distinctive feature of the PDSDBSCAN algorithm is the merging procedure that is based on a disjoint-set data structure and a tree, which allows for an effective representation of clusters and their merging. Similar to the most previously presented approaches, the parallel computation of local clusters is performed on the slave nodes.

All presented approaches so far dealt with the parallelization approaches using master-slave or shared-memory architecture. Summarized, the usual procedure to parallelize the DBSCAN algorithm is either to distribute the distance computations on several nodes and to label the objects in a master node, or to compute local clusterings in the slave nodes and to merge the partial clusterings in the master node. I.e., the master node is always the central place where the final clustering is computed. As we will see in the following section, the existing MapReduce-based techniques follow similar ideas.

5.1.2 MapReduce-based approaches

In this section, the MapReduce-based implementations of the DBSCAN algorithm are introduced.

To the best of our knowledge, there are only two approaches that deal with this task at the present point of time (state 2014): these are the MR-DBSCAN algorithm [HTL+11] by He et al. and an approach introduced by Dai and Lin [DL12] (in following abbreviated by *PartMR-DBSCAN*). Both approaches have a similar structure: after initial data partitioning by means of a d-dimensional grid, each partition (cell in the grid) is clustered by the

DBSCAN algorithm. Then, the connections between local clusterings are determined and the final global clustering is computed. Finally, the global labeling of the objects is established. As we can see, the general structure of the algorithms resembles the non-MapReduce-based approaches from previous Section 5.1.1. Due to the functional constraints of the underlying MapReduce framework and the potentially large datasets, the main differences are the utilization of a grid partitioning technique instead of tree index structures and a multi-step architecture. Similar to non-MapReduce-based approaches, MR-DBSCAN and PartMR-DBSCAN combine the local clustering results in a single master node. The experimental evaluation showed that this can become the main bottleneck of both algorithms.

The main focus of PartMR-DBSCAN is the determination of a good data partitioning that aims at reducing the number of objects that must replicated over multiple reducers as well as at a good workload balancing of the algorithm. For that, each dimension is recursively split by a slice of size 2ε in sub-partitions such that the split slice contains as few elements as possible. When such a partitioning is found, the DBSCAN algorithm is executed in every partition in parallel, and the results are combined in the subsequent step. By carrying out the computations of the local clusters in the map-phase, no communication between map- and reduce-phase takes place. That makes this approach well suited for clusters with slow networks. However, in PartMR-DBSCAN only the computation of local clusterings is performed in a parallel way. The determination of a good partitioning and the combination of the local clusters to the overall result are executed on a single master node such that the overall algorithm may perform poor on very large datasets.

The MR-DBSCAN algorithm is a grid-based approach that similarly to PartMR-DBSCAN first searches for a good data partitioning, and then determines local clustering in every partition. However, in contrast to PartMR-DBSCAN, the strategy for determination of the cell sizes exploits the marginal data histograms with a large number of bins, which are then combined to larger bins with approximately equal number of objects in a greedy way. This procedure basically equals to the greedy partitioning strategy proposed in the PHiDJ algorithm for variable grids. After determining the partitioning

in a single MapReduce job, the DBSCAN algorithm is executed in every cell which additionally contains objects from all the surrounding neighbor cells. The resulting local clusterings are then combined in a sequential manner to a global clustering and the global relabeling of the objects is performed. Similar to the PartMR-DBSCAN, the sequential global clustering computation may result in a poor performance of the algorithm on large datasets and some parameter settings.

A further approach, investigated by Fu et al. [YXF11], cannot be considered as an implementation of the original DBSCAN algorithm, due to the employed sampling procedure and the k-means-like approximation of the clustering result. For that reason, in the following we will concentrate on the MR-DBSCAN and PartMR-DBSCAN approaches.

5.2 JBMR-DBSCAN approach

In Section 2.2 (page 24), we already introduced the general idea of our novel solution JBMR-DBSCAN (*Join-Based MapReduce-DBSCAN*) and intuitively explained the single steps of the algorithm. In this section, the concrete MapReduce implementations will be presented and then evaluated in Section 5.4. To recap, the JBMR-DBSCAN algorithm consists of three steps:

1. Similarity self-join on the database objects
2. Determination of core points
3. Finding connected components with respect to $minPts$ constraint

In the following, we present the MapReduce implementations for every step.

Similarity self-join The similarity self-join can be any of the introduced techniques in Chapter 3. In our work, we mainly consider vector data, and therefore, in the experimental evaluation, the MR-DSJ algorithm was used. For following phases, let the similarity self-join graph $G_{\bowtie_\varepsilon} = (V_{\bowtie_\varepsilon}, E_{\bowtie_\varepsilon})$ (cf. Definition 2.10) be the output of the first phase of the JBMR-DBSCAN algorithm.

Core point determination The second step of the JBMR-DBSCAN is the determination of core objects in the database. To call in mind, an object o is called a core object, if $|NN_\varepsilon(o)| \geq minPts$, i.e., if o has at least $minPts$ objects in its neighborhood with smaller distance as ε. This definition corresponds to the condition that the degree of a vertex $v \in V_{\bowtie_\varepsilon}$ is larger than $minPts$. I.e., the set of core objects corresponds to all vertices $v \in V_{\bowtie_\varepsilon}$ with $deg(v) \geq minPts$. This task is easily translated into a MapReduce job. As the graph $G_{\bowtie_\varepsilon}$ is stored in the distributed file system as a set of edges (o, p), the result of two MapReduce jobs (reducer Listings 5.1 and 5.2,) is a dataset, in which every non-core object is labeled by a negative sign of the vertex ID, and all edges with a non-core object as source are removed. I.e., the result is the expected input core similarity graph $G^{core}_{\bowtie_\varepsilon}$ (cf. Definition 4.1) of the CC-MR-core algorithm (cf. Section 4.4.5).

The first MapReduce job determines the core/non-core objects, and the second one labels the non-core objects in the graph with the negative sign. The map function of the first MapReduce job is the identity mapper that simply forwards the read edges as $(key, dest)$ tuple to the reducer. The reducer of the first MapReduce job (Listings 5.1) counts the number of incoming edges with the same key vertex and either emits a pair $(key, dest)$, if the cardinality is larger than $minPts$, or it emits $(key, -key)$ pair, that indicates that key is a non-core object.

Listing 5.1: Reducer for the degree counting MapReduce job.

```
1  void reduce(int key,  Iterator<int> destinations)
2      sum = 0
3       list  = empty list
4
5      // count the degree of the key
6      for ( int dest : destinations )
7        sum += 1
8        // gather adjacent nodes if  still  not clear  if key is a core point
9        if ( sum < minPts )
10         list .add(dest)
11       else // key is a core point
```

```
12          emit(key, dest)
13
14      if ( sum < minPts ) // key is NOT a core point
15        emit(key, −key)
16      else // key IS a core point
17        for (dest in list ) emit(key, dest)
```

The map function of the second MapReduce job is again an identity mapper. By adjusting the *shuffle phase* to sort the values in an ascending order, the reducer function of the second MapReduce job (Listing 5.2) checks the sign of the first destination vertex $dest$. If $dest$ is negative, then the src vertex is a non-core point, otherwise it is a core point. Depending on whether src is a core or non-point, the edge $(dest, src)$ or $(dest, -src)$ is emitted.

Listing 5.2: Reducer for vertex labeling.

```
1 void reduce(int src, Iterator<int> destinations)
2     firstDest = destinations[1]
3     isCore = ( firstDest >= 0 ) // Is src a core/non−core point?
4
5     if (isCore)
6       for ( dest : destinations )
7         emit(dest, src)
8     else // non−core
9       emit(dest, −src)
```

Finding connected components The last step of the JBMR-DBSCAN algorithm is the determination of connected component with respect to the $minPts$ constraint, and it was already presented in Section 4.4.5. As the result of the core point detection phase is a valid core similarity self-join graph $G^{core}_{\bowtie_\varepsilon}$, the computed connected components are the correct result for the DBSCAN clustering algorithm.

5.3 Integrated JBMR-DBSCAN approach

The presented approach in Section 5.2 is a general solution for DBSCAN clustering that is applicable for all kind of metric data. However, it may suffer from exploding size of the intermediate result. For large ε values, the result of the join step may become very large, as the following example shows. Assume a small 10-dimensional dataset with 100k objects and the ε value that result in $5\%|DB^2|$ large similarity graph output. Then, the join phase generates a 3815 Mb large output similarity graph out of the 3.8 Mb large input database! This corresponds to an around 1000-times increase of the data amount that have to be processed in the connected components phase.

For that reason, in the following, we investigate solutions that integrate the similarity self-join phase and the connected component phase in order to reduce the intermediate result size and that way improve the overall performance of the our JBMR-DBSCAN algorithm. The basic idea is to perform the local connected component search during the join phase and to output local star-graph components instead of complete (potentially very dense) similarity subgraphs in every reducer. In general, this idea is applicable for every existing similarity self-join algorithm as one can store (part of) the generated subresult of a single reducer, perform the connected component search with respect to core and non-core objects on the stored subresult, and then output the reduced graph only.

However, to produce good results, the objects of the computed join-subresult in a single reducer must be located in the same data space regions; otherwise, no large subcomponents can be identified and no data reduction is possible. For that reason, in this thesis, we concentrate on a solution for the MR-DSJ algorithm as the underlying grid approach ensures the data locality in a single reducer.

5.3.1 Basic procedure

To recap, the basic idea of our integrated approach is to precompute local connected components in the MR-DSJ reducers and to output the resulting star-like graphs along with some additional information instead of the full local similarity self-join intermediate results. The following steps describe the re-engineered workflow of the JBMR-DBSCAN algorithm:

1. Each reducer in the adjusted MR-DSJ algorithm:
   - stores a partial intermediate result of the join
   - determines core and non-core objects based on the stored subresult
   - emit the found connected components + additional information described later on
2. In the subsequent *CorePts* job, the core and non-core objects are identified
3. CC-MR-core algorithm is applied on the core similarity graph from previous step

To apply this algorithmic scheme, two problems have to be addressed. First, as the intermediate results of single reducers in the join step can become very large, a meaningful selection strategy of the join result that leads to the largest data reduction is necessary. In this thesis, we propose several strategies in Section 5.3.1.2 and evaluate them in Section 5.4.

The second problem concerns the determination of core and non-core objects in a single reducer. As in general a single reducer r does not process the complete dataset, some objects from r may have join partners in other reducers such that a global determination of core and non-core objects inside a single reducer is not possible. To take account of this problem, we enrich the generated local subresults of every reducer by additional information using the coding scheme introduced in Section 5.3.1.1 and present an algorithm for global determination of the core and non-core objects from these local results.

I.e., our main contributions are:

- An extended MR-DSJ reducer with local core/non-core point determination and local connected component computation
- A *CorePts* algorithm for determining global core/non-core objects in the graph

Next Section 5.3.1.1 describes the coding scheme and the *CorePts* algorithm. The storage strategies for the join subresults are introduced in Section 5.3.1.2.

5.3.1.1 Coding scheme and CorePts algorithm

Recall that an object is called a core point if it has at least $minPts$ objects in its neighborhood. In other words, if the similarity graph of the database is known, an object is a core point if it has at least $minPts$ adjacent vertices. As a single reducer in the MR-DSJ algorithm computes a subgraph of the overall similarity graph and therefore does not contain edges to vertices from other reducers, it is not able to identify all core objects, as the observed vertex degrees of some core objects may be to small. To tackle this problem, we extend the output of each MR-DSJ reducer for each vertex v by the local degree of v (i.e., the number of neighbors of v) and use this information in the subsequent step to compute the global degree of v by summing up the local degrees. Technically, the algorithm emits an additional edge $(v, -(\#\text{neighbors}))$ for every vertex v of the reducer. Here, we implicitly assume that no edge will be counted multiple times in different reducers. To guarantee this, all proposed strategies rely on the bitcode pruning of the MR-DSJ algorithm that on its side is proven to be inherently duplicate free.

Provided local degree information of the vertices, the two-phase $CorePts$ algorithm determines global core and non-core objects for the subsequent CC-MR-core algorithm. The first phase constitutes an identity mapper and the reduce function in Listing 5.3.

Listing 5.3: Phase 1: Core/non-core object determination.

```
1 void reduce(int src, Iterator<int> dests)
```

```
2     int count = 0
3     foreach dest in dests
4       if ( dest < 0 ) // local degree of src in some reducer
5         count += dest // accumulate local degrees of src
6       else emit(src, dest) // process regular edges
7
8       if ( −count < minPts ) // is source core or non−core object?
9         emit(src, −src) // label non−core objects for the next phase
```

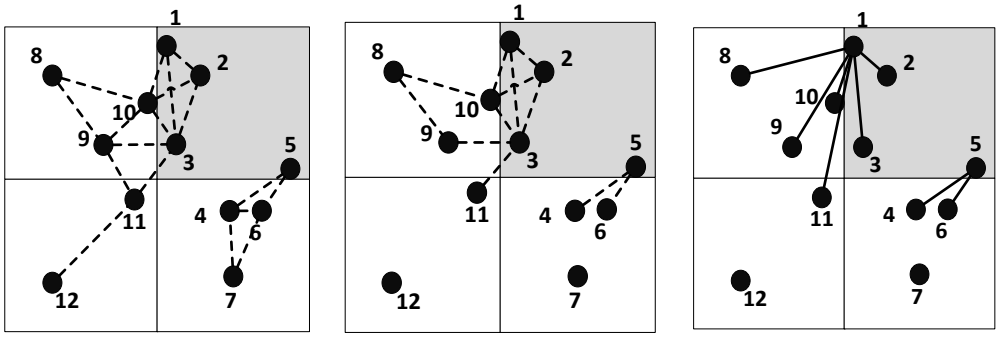
Applying the secondary sort facility of MapReduce to sort the values of the reducer in the ascending order, the first $CorePts$-phase accumulate determined local degrees of the vertices by summing up negative values and emits a $(v, -v)$ edge for every non-core object v. This step strongly corresponds to the degree counting job (cf. Listing 5.1) with some minor changes needed to deal with regular edges. The second $CorePts$-phase corresponds to the vertex labeling job in Listing 5.2 and remains completely unchanged.

5.3.1.2 Storage strategies

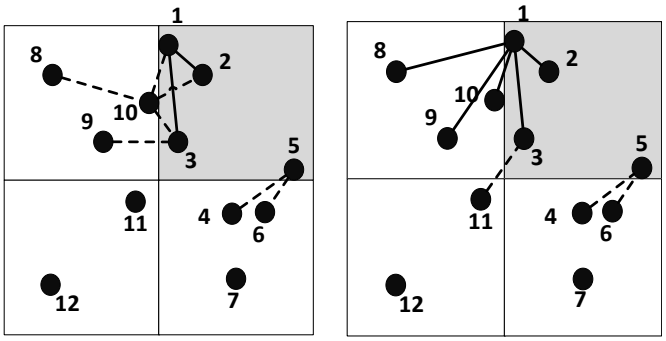
To recap, the basic idea of the integrated approach is to perform local connected component search in the join phase in each reducer and to output the condensed graphs instead of the complete join graph. To perform this search, parts of the join results of a reducer must be locally stored in the reducer. Intuitively, the more join pairs are stored the smaller will be the condensed output, since potentially larger components will be found. In this section, we describe and evaluate three storage strategies, i.e., strategies which of the join pairs to store. These strategies mainly differ in the number of stored edges and share the property that only those edges are stored that were not pruned by the bitcode pruning. The main reason for this decision is the potentially exponential increase of the distance computations accompanied by strategies in which the duplicate edges are not pruned and therefore have to be stored and preprocessed in the subsequent steps.

The subfigures in Figure 5.1 illustrate the outcomes of the local connected component search that only considers the join-pairs stored following the pro-

posed strategies. In all examples, we assume the $minPts$ parameter set to value 2. The four cells represent the input of some reducer, where the upper right cell is the home cell. Figure 5.1(a) depicts the full similarity join graph that the reducer would produce if no bitcode filtering is applied, and we use it for comparison with other cases.



(a) Full join result. (b) Result of the original MR-DSJ reducer. (c) *all-join-pairs* strategy.



(d) *home-cell* strategy. (e) *half-cells* strategy.

Figure 5.1: Different storage strategies and the output of the integrated JBMR-DBSCAN approach. Dashed edges are not influenced edges of the MR-DSJ algorithm. Solid edges are edges that were processed and possibly adjusted by the algorithm.

Dashed edges in the graphs decodes such edges that are not involved into computation of the connected components and that they are emitted as in the original CC-MR algorithm. Solid edges are the adjusted edges (e.g. newly created or in some other way touched) that are then emitted after determi-

nation of connected components. Following this coding scheme, the output of the original MR-DSJ algorithm in Figure 5.1(b) consist of (dashed) edges that connect the objects of the home cell with objects from the neighboring cells.

***all-join-pairs* strategy** The *all-join-pairs* is the most straightforward solution that stores the complete join result of a reducer of the original MR-DSJ algorithm. Figure 5.1(c) illustrate the outcome of the local connected component search for this strategy. The both components from the original MR-DSJ join output (Figure 5.1(b)) are transformed into two star-graphs with the central vertices 1 and 5. Due to $minPts = 2$, the center of the smaller component is the node 5, since with respect to the stored edges the node 4 is considered to be a border object

As one recognizes, even in this simple example the number of edges is reduced from 12 to 8. As we will see later on, this technique achieves significant reduction of the output in real examples. Please note that due to the proposed coding scheme, all strategies emit additional edges $(v, -(\#\text{neighbors}))$ with the node degrees (cf. Section 5.3.1.1), which increase the overall communication overhead but are not counted in this example. If we consider those edges, the final result becomes larger than the full join output. However, as we will show in the experimental section (Figures 5.3 and 5.4, page 153), in real datasets with many more edges, the achieved savings from connected component phase are significantly larger than the additionally emitted edges.

***home-cell* strategy** The advantage of the *all-join-pairs* strategy is its ability to substantially reduce the amount of emitted data. However, for dense datasets, the intermediate join result may become extremely large and exceed the limits of the physical memory such that it cannot be stored in the reducer node. The *home-cell* strategy tackles this problem by storing the join result of the home cell only and performs the local component search on this intermediate result. Figure 5.1(d) depicts this case. In the example, only the edges between the nodes 1, 2, 3 and 5 are considered during the connected

component determination. Therefore, only the single edge $(2, 3)$ is removed from the final result.

***half-cells* strategy** The *home-cell* and *all-join-pairs* represent two extremes, the first one stores as less as possible and the second one as many as possible intermediate results. The *half-cells* strategy is the compromise between these solutions; it stores the intermediate results between cells stored in a single MR-DSJ reducer and the home cell, i.e., from all cells sharing the largest bit in their bitcodes with the home-cell. Figure 5.1(e) illustrate the result of this strategy. The output of *half-cells* mainly resembles the output of the *all-join-pairs* strategy with the difference that no edges from the bottom cells are considered during the connected component determination. Therefore, the edge $(3, 11)$ is not forwarded to the central vertex 1 of the component. In this example, the *half-cells* achieves the same reduction as the *all-join-pairs* and also in larger datasets in the experimental evaluation, it often achieves comparable performance.

5.3.2 Implementation in MapReduce

In the previous section, we described the central concepts needed to extend the MR-DSJ algorithm and presented the MapReduce implementation of the $CorePts$ algorithm. Now, we provide the implementation details of the adjusted MR-DSJ algorithm that along with the CC-MR-core algorithm then completes our integrated JBMR-DBSCAN algorithm. To integrate the local connected component search and local core/non-core point determination, only the reducer of the MR-DSJ algorithm must be modified. The necessary changes are provided as a high-level pseudo code in Listing 5.4. We use following notation in the source code: G_V and G_E denote the vertex set and edge set of a graph G, respectively. *degree(u)* is the degree of a vertex u. Finally, letter *E* alone denotes a set of edges.

Listing 5.4: Modified reducer of MR-DSJ algorithm for the integrated JBMR-DBSCAN algorithm.

```
1  void reduce(int partition_id , Iterator<Point> values, StorageStrategy strategy)
2
3      // 1. perform original MR-DSJ but instead of emitting the results ,
4      // store them in a similarity graph constrained to the applied
5      // storage strategy (parameter 'strategy' of the function)
6      // and in otherVertices (not stored edges).
7      (storedSimGraph, otherEdges) = computeSimJoin(id, values, strategy)
8
9      // Emit local vertex degree for not stored edges
10     foreach (u, v) ∈ otherEdges
11       emit( (u, −1) )
12       emit( (v, −1) )
13
14     // Emit the degree of the stored edges
15     foreach u ∈ storedSimGraph_V
16       emit( (u, −degree(u)) )
17
18     // 2. determine and label local core and non−core points
19     labeledSimGraph = labelVertices(storedSimGraph)
20
21     // 3. find local connected components using only core objects
22     // in labeledSimGraph. The edges between non−core objects remain
23     // unchanged!
24      list <E> subcomponents = det_conn_comps(labeledSimGraph)
25
26     // 4. Output the determined connected components
27     foreach E ∈ subcomponents
28       emit( (u, v) ∈ E )
29       emit( (v, u) ∈ E )
30
31 // compute connected components and return a list in which each element
32 // is a set of edges of a single component
33  list <E> det_conn_comps(labeledSimGraph)
34     // 1. determine connected components in form of star graphs
35     // for core objects only
```

```
36    connCompsGraph = CCMR_local(labeledSimGraph_cores)
37
38    // 2. Insert the edges containing the non−core objects
39    // to the connected components
40    connCompsGraph = insertEdges(E_non−cores)
41
42    // 3. Remove edges between non−core objects that share
43    // at least one core object
44    foreach edge (u, v) ∈ connCompsGraph
45      if ( !is_core(u) && !is_core(v) )
46        if ( ∃w ∈ connCompsGraph_V : is_core(w) &&
              (u, w) ∈ connCompsGraph_E && (v, w) ∈ connCompsGraph_E )
47          connCompsGraph_E = connCompsGraph_E \ (u, v)
48
49    // return connCompsGraph_E as edge set for each component
50    foreach edge (u, v) ∈ connCompsGraph_E
51      emit( (u, v) )
52      emit( (v, u) )
```

First step in line 7 is basically the original MR-DSJ procedure for computing the similarity self-join result. The difference is that the resulting pairs are not emitted but stored for further postprocessing. In lines 10–16, the local vertex degrees are emitted for the subsequent $CorePts$ algorithm. Since for the not stored pairs no adjacency relationship is given, for each vertex the value -1 is emitted. For the stored edges, we emit the number of adjacent vertices *before* connected components search, such that no information will be lost in the following core/non-core point determination step. In the next step in line 19, the local core and non-core objects in the stored similarity graph $storedSimGraph$ are identified. The obtained core objects are then used for computing the local connected components in form of star graphs (line 36) for example by executing a local version of the CCMR algorithm. In the next step, the non-core objects with belonging edges are reinserted into the star-graphs in line 40. At this point, the number of join pairs is already significantly reduced, as all connections between core objects (except

for the forward and backward edges to/from the center of the component) are removed. However, the number of edges can be reduced even more, if all edges between non-core objects connected to the same core object are removed. This further edge reduction is performed in lines 44 – 47. Please note that we left out this optimization in the example in Figure 5.1 to keep it simple. After this step, the final result is written onto distributed file system in lines 27–28.

5.3.3 Proof of correctness

In this section, we provide the proof for the correctness of the proposed extension of the JBMR-DBSCAN algorithm. Intuitively, the following Theorem 5.1 proves that if a vertex p belongs to a component with a center vertex z in the original similarity self-join graph, then it will belong to the same component with center z in a graph constructed through application of the CorePts-job on the union of reduced subgraphs. This means, the provided proof deals with all three storage strategies from Section 5.3.1.2, as it does not differentiate from how many cells the join result is stored but only proves that the changes of the local similarity graphs do not falsify the overall result. Technically, we will show that the output of the integrated approach does not cut connections between any two core objects p and q. This does not mean that the existing paths between p and q remain unchanged, in contrary, the goal of the integrated approach is to remove as many edges as possible. However, if p and q are density connected in the similarity graph $G_{\bowtie_\varepsilon}$, then they are also density connected in the transformed graph produced by the integrated approach.

Theorem 5.1 *Let $G_1, \cdots, G_m$, $G_i \in G_{\bowtie_\varepsilon}$ for all i, be induced subgraphs of the similarity self-join graph $G_{\bowtie_\varepsilon}$ with $V_i \subseteq V_{G_{\bowtie_\varepsilon}}$, $E_i \subseteq E_{G_{\bowtie_\varepsilon}}$ and $E_i \cap E_j = \emptyset$ for all i, j computed by m reducers. Further, let $z_1, \cdots, z_k$ be k centers of the connected components with respect to $minPts$, i.e., the centers of the k clusters of the dataset, and let $C^G(z_i) \subseteq V_G$ denote the vertex set of a single component with center z_i computed from a graph $G = (V, E)$. Then it holds: if*

$p \in C^{G_{\bowtie_\varepsilon}}(z)$ then $p \in C^{CorePts\left(\bigcup_j G'_j\right)}(z)$, where G'_j denotes a graph computed by the integrated JBMR-DBSCAN approach in the reducer r_j, and $CorePts(G)$ is the output of the CorePts-job as described in Section 5.3.1.1.

Proof 5.1

***Case 1**:*

Let $z, p \in V_{G_j}$, i.e., both vertices are in the vertex set of some reducer r_j, and let both vertices be density connected by a series of core objects $p_1, \cdots, p_n$, $p_i \in V_{G_j}$. Then, as the integrated JBMR-DBSCAN employs the CC-MR-core algorithm, vertex p will be obviously assigned to the vertex z.

***Case 2**:*

Let $z, p \in V_{G_j}$, i.e., both vertices are in the vertex set of some reducer r_j, but now z and p are <u>not</u> density connected in G_j, i.e, in the series of objects $p_1, \cdots, p_n$ there is at least one non-core point p_i, or an edge between some p_i and p_j is missing. In both cases, the reducer r_j is not able to connect p and z. However, the global CC-MR-core algorithm will be able to assign p to z after the $CorePts$ step as we now prove.

***Case a:** p_i is non-core point*

In this case, the integrated JBMR-DBSCAN emits the edges $(p_i, -n_j)$ in every reducer containing p_i, where n_j is the degree of p_i in reducer r_j, and all edges from p_i to its neighbors. The subsequent $CorePts$ phase combines partial degrees n_j and will determine that p_i is a core point. Since all forward and backward edges from p_i to the neighbors are kept, the global CC-MR-core algorithm will eventually connect the vertices p and z.

***Case b:** Edge (p_i, p_j) is missing*

Due to correctness of the MR-DSJ algorithm, the edge (p_i, p_j) will be generated in some other reducer r_k. If in r_k the vertex p_i is a non-core, then the reducer will emit the forward and backward edges (p_i, p_j) and (p_j, p_i) such that the global CC-MR-core algorithm will eventually be able to connect z and p. If in reducer r_k both p_i and p_j are assigned to some other central vertex y, then the forward and backward edges $(p_i, y), (y, p_i)$ and $(p_j, y), (y, p_j)$ will be generated. As we assumed that p_j and p, and p_i and z are density connected, the edges (z, p_i), (p_i, z), and (y, p), (p, y) will be generated, too. Therefore there is a path from z

to p over (p_i, y) and (y, p), such that the global CC-MR-core will assign p to z.
***Case 3**:*
Now let z and p be the output of tho different reducers. Then the proof pretty much follows the proof of Case 2. As for core and non-core objects, the algorithm generates forward and backward edges, no edge will be missing in the global output, such that the $CorePts$ algorithm will be able to identify all core points and the global CC-MR-core will eventually connect all vertices to the appropriate centers of the components.

■

5.4 Evaluation

In this section, the JBMR-DBSCAN approach is compared with the competitor approaches MR-DBSCAN and PartMR-DBSCAN. In the first part of this evaluation (Section 5.4.1), the JBMR-DBSCAN and the integrated JBMR-DBSCAN approaches are considered. Afterwards, the best solution is compared with the competitor algorithms in Section 5.4.2. The measured quantities are overall runtime and the overall communication of the approaches. In the experiments, we use several synthetic and real world databases described later on.

Hardware and software The initial comparison of the JBMR-DBSCAN and its integrated version is conducted on a cluster with 4 nodes running as virtual machines on a high-end server with Cloudera Hadoop CDH 4.7. Each node is equipped with a 2-core Intel(R) Xeon(R) CPU X5650 with 2.67 GHz. The master node have 5 GB RAM, the slave nodes 3 GB at their disposal. The storage capacity of the distributed file system is 180 GB with a 3 times replication.

The final evaluations are conducted on the 14-node Hadoop cluster running Cloudera Hadoop CDH3b4 described in previous sections. All approaches were implemented in Java 6.

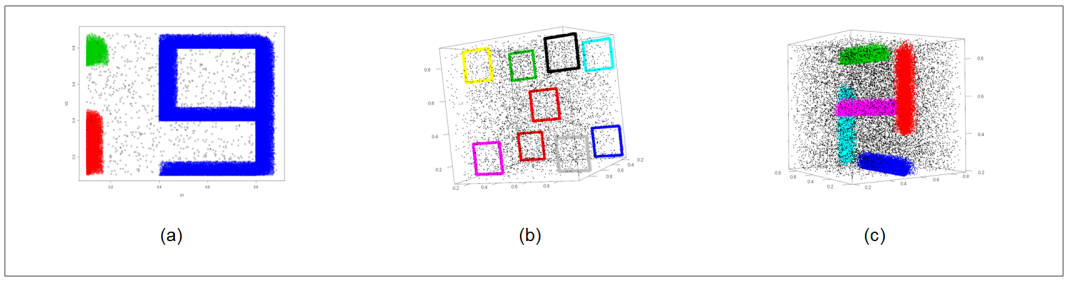


Figure 5.2: Visualization of synthetic 2d, 3d, and 4d datasets.

Datasets In our experiments, we use five datasets, consisting of four synthetic datasets with varying dimensionality and a real-world minutiae dataset. The properties of all datasets are summarized in Table 5.1. Figure 5.2* de-

Name	Abbreviation	Dimensionality	# objects
2d_i9	2d	2	2 mio
3d_blocks	3d	3	500k
4d_lines	4d	4	1 mio
5d_lines	5d	5	1 mio
minutiae	minutiae	3	11 mio

Table 5.1: Properties of the used datasets.

picts the samples from the 2d, 3d and projected 4d datasets. As one recognizes, in these datasets we vary the shapes and the densities distributions to investigate the influence of these properties on the performance of the algorithms. The 4- and 5-dimensional datasets consist of clusters in form of non-overlapping lines, while the 3-dimensional datasets consists of 9 non-overlapping block-shaped clusters. The values in all synthetic datasets are normalized to the [0,1] range.

The "minutiae" dataset from the "NIST Special Database 14†/29‡" contains 3-dimensional coordinates of fingerprint minutiae and quality values, distributed in the ranges [0, 832], [0,768] and [0,100].

*Image from [Gra13].
†`http://www.nist.gov/srd/nistsd14.cfm`
‡`http://www.nist.gov/srd/nistsd29.cfm`

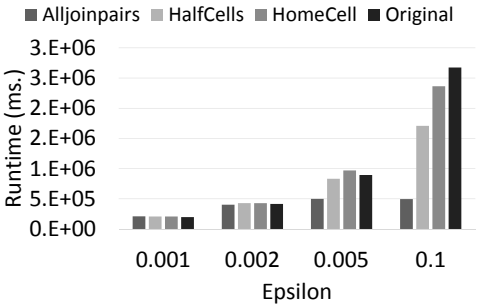
5.4.1 Evaluating JBMR-DBSCAN

In this section, we compare the JBMR-DBSCAN algorithm with the integrated approach with the *all-join-pairs*, *home-cell* and *half-cells* storage strategies described in Section 5.3.1.2. The experiments are conducted on 500k large samples of all synthetic datasets and the minutiae dataset. The obtained running times and overall communication of all approaches are depicted in Figures 5.3 and 5.4, respectively. The missing values are due to algorithm failures, either because of not sufficient storage capacities on the distributed file system or due to exceeded resources on the single computing nodes. The excess of the storage capacities arose from the too large global intermediate output of the join phase while the failures of single nodes is attributable to the too large intermediate output of a single reducer node.

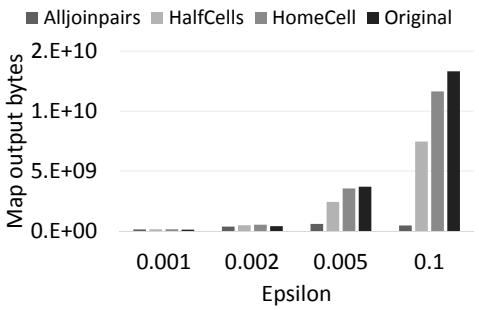
The names *Alljoinpairs*, *HalfCells* and *HomeCell* in the legend correspond to the appropriate storage strategies of the integrated approaches, *Original* stands for the non-integrated JBMR-DBSCAN algorithm. The results show a clear superiority of the *all-join-pairs* strategy over all other approaches in almost every case. The reduction of the emitted result pairs positively influence the running time of the overall algorithm as Figure 5.5 exemplary shows for the 3d dataset for $\varepsilon = 0.01$. The almost equal runtime of the join phase (Join) is followed by significantly lower runtimes in the core point detection (CorePts) and connected component search (CCMR) phases. The similar behavior can be observed in all other datasets.

In general, the experimental result show that the size of the intermediate result is strongly correlated with the overall running time. In the performed experiments, the intermediate result size was even the primary factor for the overall runtime.

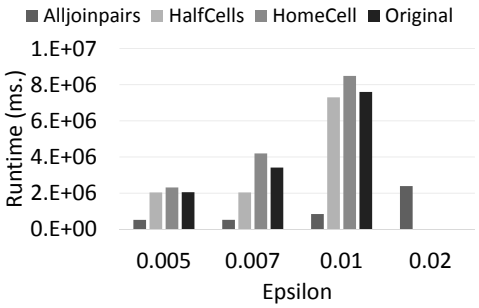
Interestingly, the evaluation also shows that not every storage strategy was able to outperform the JBMR-DBSCAN approach in terms of intermediate result size. E.g., for the 3d dataset, the communication of the *home-cell* strategy was constantly larger than the one of JBMR-DBSCAN for all ε values. Obviously, the additional information emitted for the subsequent core point detection phase was larger than the savings from the integrated lo-



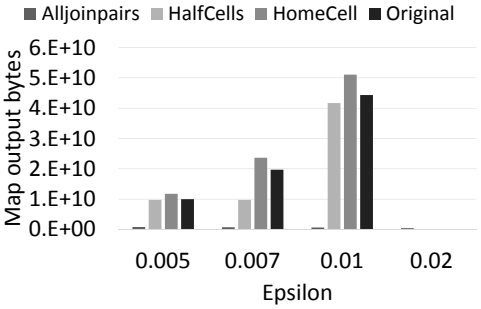
(a) Runtime (2d).



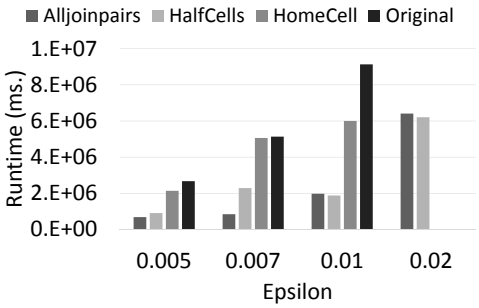
(b) Communication (2d).



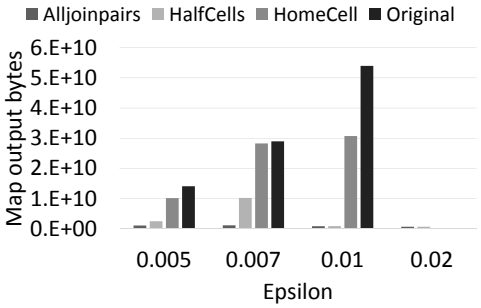
(c) Runtime (3d).



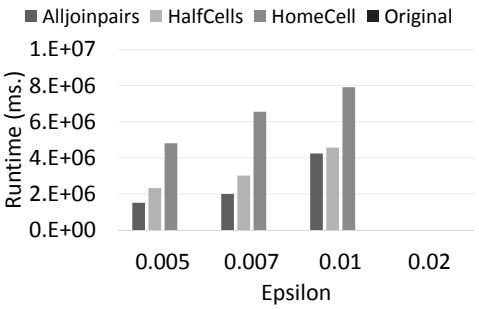
(d) Communication (3d).



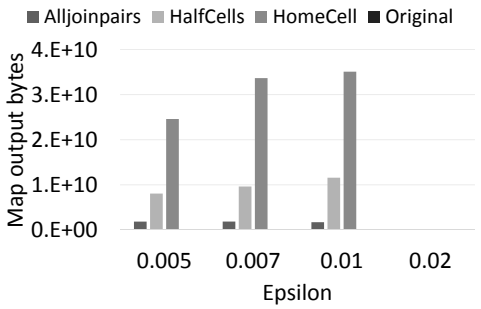
(e) Runtime (4d).



(f) Communication (4d).

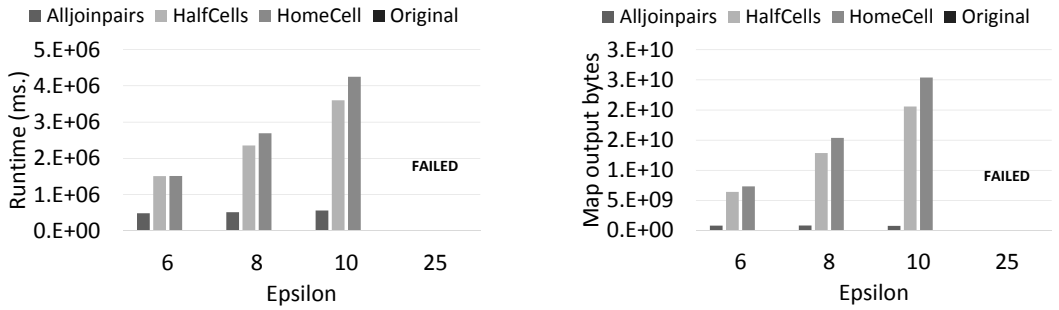


(g) Runtime (5d).



(h) Communication (5d).

Figure 5.3: Comparison of JBMR-DBSCAN algorithm and the integrated approaches on the synthetic datasets (2d, 3d, 4d, 5d).



(a) Runtime (minutiae). (b) Communication (minutiae).

Figure 5.4: Comparison of JBMR-DBSCAN algorithm and the integrated approaches on the minutiae dataset.

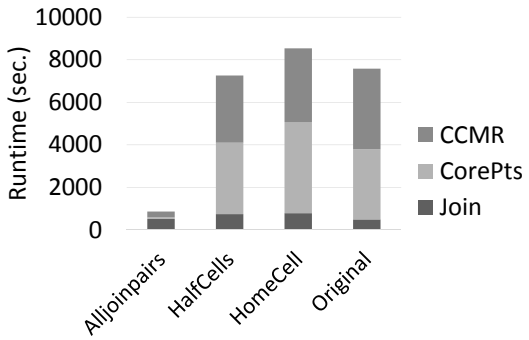


Figure 5.5: Running times of every phase on the 3d dataset with $\varepsilon = 0.01$.

cal connected component search. However, the *all-join-pairs* and *half-cells* constantly outperformed the JBMR-DBSCAN in terms of communication and runtime.

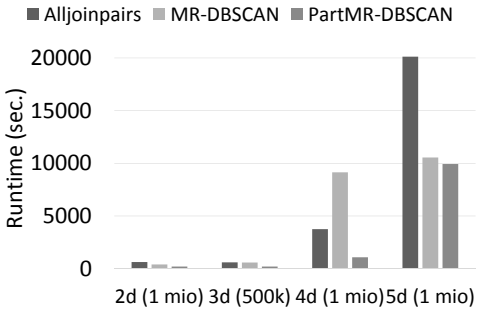
Due to the superiority of the *all-join-pairs* over other approaches, we will use this strategy for the upcoming experiments.

5.4.2 Evaluating JBMR-DBSCAN and competitive approaches

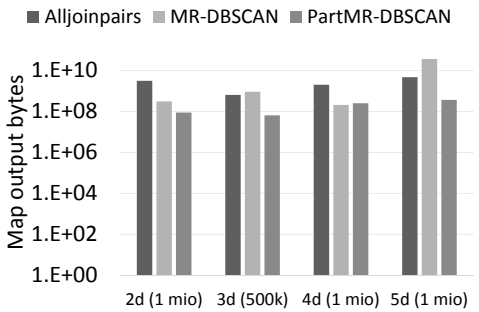
In this section, we compare our integrated JBMR-DBSCAN with *all-join-pairs* storage strategy with the competitive techniques MR-DBSCAN and PartMR-DBSCAN. Table 5.2 summarizes the used settings for the experiments. Figures 5.6(a) and 5.6(b) show the running times and the overall communication (mapper output bytes) on the synthetic datasets. In the used datasets, the

Dataset	ε	minPts	Dataset size	number reducers
2d_i9	0.002	50	1 mio	112
3d_blocks	0.1	30	500 k	112
4d_lines	0.1	30	1 mio	112
5d_lines	0.005	100	1 mio	112
minutiae	8	50	5 mio	112

Table 5.2: Used parameter settings for the experimental evaluation on synthetic datasets.



(a) Running time of all algorithms.



(b) Running time of all algorithms.

Figure 5.6: Comparison of the integrated JBMR-DBSCAN approaches with *all-join-pairs* storage strategy vs. competitor algorithms.

PartMR-DBSCAN algorithm clearly outperforms the MR-DBSCAN algorithm as well as our JBMR-DBSCAN approach, in terms of the overall running time and the produced communication. The MR-DBSCAN and the JBMR-DBSCAN algorithm share the second place; depending on the used dataset and parameter setting the first or the latter algorithm perform better.

This result appears very disappointing at first sight as even our most efficient proposed strategy is significantly slower and does produce much larger output. However, the JBMR-DBSCAN approach possesses several features that make it applicable in situations for which the competitor algorithms are either not designed for or in which they will perform poorly.

The first distinctive feature of the pure JBMR-DBSCAN algorithm is its ability to process metric data if some existing metric similarity self-join approach instead of our MR-DSJ is used. The MR-DBSCAN and PartMR-DBSCAN algorithms base on the grid partitioning and therefore require vector rep-

resentation of the data. Certainly, applying pivoting technique, the metric objects can be transformed into vector data such that after careful adjustment of the both techniques they would be applicable for metric spaces, too. This, however, would significantly reduce the efficiency of both approaches, as the employed grid techniques of MR-DBSCAN and PartMR-DBSCAN algorithms lead to a 3^d times data replication and therefore only a few number of pivot objects could be used. As the selectivity of the lower bound distance function $L_\infty(\cdot,\cdot)$ strongly depends on the number of pivots objects, the MR-DBSCAN and PartMR-DBSCAN would have to perform significantly more distance computations as for example the JBMR-DBSCAN approach connected with our PHiDJ algorithm.

Further, the efficient connected component search phase of the JBMR-DBSCAN and its integrated approach is superior to the cluster merging techniques used in the MR-DBSCAN and PartMR-DBSCAN algorithms when large amount of data have to be merged. To see that this point is relevant, consider Table 5.3 that depicts the running times of the single phases of all algorithms for synthetic and the minutiae dataset. The presented values clearly show the strong and weak sides of the algorithms. For the synthetic datasets, we see that the most consuming parts of the algorithms is the local determination of the DBSCAN clustering results. In the case of the JBMR-DBSCAN algorithm, this phase is even responsible for algorithm failure on the real minutiae dataset. At the same time, the MR-DBSCAN and PartMR-DBSCAN were able to compute the partial DBSCAN results on the same minutiae datasets but were not able to merge the clustering results, as the number of produced local clusters was too large for processing on a single workstation. At the same time, our CC-MR algorithm easily copes with datasets with millions objects and billions edges such that it could compute the result if we would exchange the employed similarity self-join algorithm.

PartMR-DBSCAN	**2d**	**3d**	**4d**	**5d**	**minutiae**
Grid partitioning	88	17	56	147	342
Local DBSCAN	102	176	1019	9769	1610
Cluster merging	1	0.5	0.5	4	**FAILED**
Postprocessing	6	1	3	12	-
Sum	*197*	*194.5*	*1078.5*	*9972*	-
MR-DBSCAN	**2d**	**3d**	**4d**	**5d**	**minutiae**
Grid partitioning	20	31	19	23	28
Local DBSCAN	338	469	8906	9839	919
Core point determination	33	70	211	663	194
Cluster merging	0.6	4	0.4	8	**FAILED**
Postprocessing	7	4	4	15	-
Sum	*398.6*	*578*	*9140.4*	*10548*	-
JBMR-DBSCAN (*all-join-pairs*)	**2d**	**3d**	**4d**	**5d**	**minutiae**
Self-join/local component search	195	233	3273	19536	**FAILED**
Core point determination	89	76	88	128	-
Cluster merging	340	290	386	457	-
Sum	*624*	*599*	*3747*	*20121*	-

Table 5.3: Running times (rounded to seconds) of single phases of all algorithms.

5.5 Summary and conclusion

In this chapter, we dealt with the question how to parallelize the DBSCAN algorithm in the MapReduce framework. We proposed a multi-step architecture consisting of similarity self-join, core point detection and connected component phases and presented novel solutions to efficiently cope with this task. Finally, we compared our JBMR-DBSCAN approach with existing approaches in the literature.

The experimental evaluation showed that the proposed solutions were not able to outperform the PartMR-DBSCAN algorithm on the employed datasets. However, as we argued, the generality of the JBMR-DBSCAN algorithm and the parallelization of all steps is a huge advantage over the competitive approaches. It not only can process metric objects but is also applicable in cases when the intermediate local DBSCAN clustering results are

very large and the usage of sequential algorithms is not possible any more. In next section, we discuss a combination of all approaches that could further improve the performance of the parallelized DBSCAN algorithm.

The main problem of the JBMR-DBSCAN algorithm is the huge output of the join phase that has to be processed in the subsequent steps. The proposed integrated approach leads to a significant improvement, but still suffers from large intermediate results. Therefore, in order to further accelerate the JBMR-DBSCAN approach, a significant reduction of the intermediate result should be the main focus of the research.

5.6 Future work

Obviously, as the DBSCAN algorithm is decomposable in independent components, the combination of most efficient techniques should gain the best overall results such that there are a large number of improvements that are worth of investigation. Here, we only present some of these possibilities.

A straightforward solution is to combine the most effective parts of the presented approaches. First part could be the partitioning strategy of the PartMR-DBSCAN that explicitly reduces to communication of the overall algorithm. Then, while computation of the DBSCAN algorithm, the bitcode pruning introduced in the MR-DSJ algorithm could be utilized to further minimize the number of distance computations and to reduce the data replication. Finally, the CC-MR algorithm can merge the local clustering results for large datasets. A further interesting extension for high-dimensional spaces would be the incorporation of the partitioning strategy from PartMR-DBSCAN into the PHiDJ algorithm.

For large datasets, the combination of the MapReduce solution with GPU based approaches will gain further acceleration of the overall running time. As the computations of the local clusterings is one of the most expensive steps of the entire algorithm, the speedup of this part will significantly increase the performance of the overall algorithm.

Part II

Projected clustering on massive data

The trend of the last years to gather more and more data in all areas of our everyday life does not only mirror in huge datasets but also in a fast growing number of attributes. The connection of multiple information sources results in datasets with dozens, hundreds and even thousands dimensions, and in some scientific areas (as for example biology) the development of sensor technology leads to data having millions of attributes. A meaningful analysis of such datasets requires sophisticated data mining techniques as for example subspace or projected clustering. The basic idea of these techniques is not only to determine the meaningful object groups, but also to identify relevant subspace in which these objects reside.
As we will discuss later on, at the present point of time, there are almost no projected clustering algorithms that are able to deal with huge amounts of data. This observation motivates this chapter, in which several extensions and improvements of the state-of-the-art projected clustering algorithm P3C are investigated, which make this algorithm applicable on large datasets in high-dimensional spaces. We show that the original model of the P3C algorithm is not suitable to deal with huge datasets. Therefore, we propose the necessary changes of the underlying clustering model and then present an efficient MapReduce-based implementation - our novel P3C$^+$-MR algorithm. The effectiveness of the proposed changes on large datasets and the efficiency of the P3C$^+$-MR algorithm are comprehensively evaluated on synthetic and real-world datasets. Additionally, we propose the P3C$^+$-MR-Light algorithm, a simplified version of P3C$^+$-MR that shows extraordinary good results in terms of runtime and result quality on large datasets. In the end, we compare our solutions to existing approaches.

Before we proceed to the technical part of this chapter, in Section 6.1, we briefly introduce the problems arising in high-dimensional spaces and make the clustering in full spaces a very challenging task. In the same section, we summarize the contributions of our work and provide the structure of the chapter.

Chapter 6

MR-P3C

Contents

6.1 Introduction **164**

6.2 Related Work **167**

6.3 Original P3C algorithm **169**

6.3.1 Definitions and notation 170

6.3.2 P3C algorithm description 170

6.4 Boosting P3C model for Big Data **175**

6.4.1 Statistical issues when dealing with Big Data . . 175

6.4.2 Improvement of the algorithmic design 179

6.5 P3C$^+$-MR in MapReduce **182**

6.5.1 Histogram building 183

6.5.2 Relevant intervals 183

6.5.3 Cluster core generation 183

6.5.4 EM 187

6.5.5 Outlier detection 188

6.5.6 Attribute Inspection 189

6.5.7 Interval Tightening 189

6.6 P3C$^+$-MR-Light algorithm **189**

6.7 Experiments **192**
6.7.1 Dataset description 192
6.7.2 Used measures 193
6.7.3 Parameter settings 193
6.7.4 Evaluation of the P3C$^+$ model 193
6.7.5 Evaluation on large synthetic datasets 197
6.7.6 Real world data 202
6.8 Conclusion **202**
6.9 Future work: Steering the P3C$^+$ algorithm **203**

6.1 Introduction

In the last decades, a large number of clustering approaches was developed. Expectation Maximization (EM) algorithm, DBSCAN and k-means algorithms are only a very small subset of existing approaches that are used in different applications. Most of these traditional approaches consider all data attributes for determining an appropriate grouping of objects. However, as recent developments show, clustering becomes less and less meaningful with a growing number of dimensions due to the so-called *curse of dimensionality*. Intuitively, curse of dimensionality states that with a growing number of dimensions the distances between objects become more and more alike, such that no meaningful grouping is possible any more. Formally, Theorem 6.1 [LV10] states that with growing number of dimensions d, the mean (i.e. expected distance between two objects) of the Euclidean norm grows proportionally to $\sqrt{d}$ but the variance remains almost constant for sufficiently large d.

Theorem 6.1 *Let Y be a d-dimensional random variable $(y_1, \cdots, y_d)$ where each component y_i is independent and identically distributed. Then, the mean $\mu_{\|Y\|}$ and the variance $\sigma_{\|Y\|}$ of the Euclidean norm are:*

$$\mu_{\|Y\|} = \sqrt{ad - b} + O(d^{-1})$$

and

$$\sigma_{\|Y\|} = b + O(d^{-\frac{1}{2}}),$$

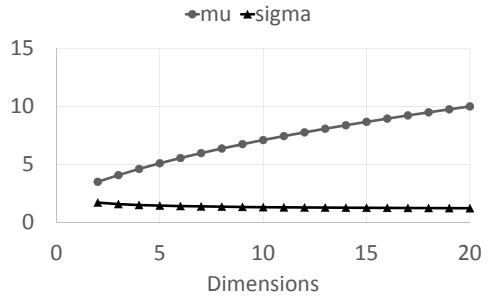
where a and b are parameters depending only on the central moments of order 1, 2, 3, and 4. The n-th central moment is defined as: $\mu_n = E[(Y - E(Y))^n]$, where $E(X)$ is the expectation of X.

Figure 6.1(a) shows the means and variances for growing number of dimensions for some fixed values a and b. As one recognizes, the variance (sigma) stagnates at the value ≈ 1, while the mean (mu) is growing the more dimensions are considered.

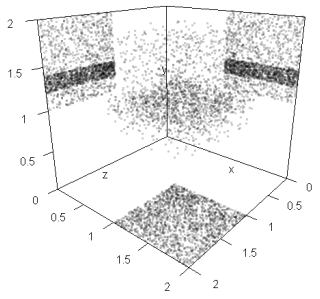
For clustering, the curse of dimensionality effect also mirrors in problems related to irrelevant or noisy dimensions. Assuming that a meaningful cluster produced by a clustering algorithm summarizes objects that represent some concept in the data, an irrelevant or noisy dimension contains information that is independent from this concept. Noise dimensions lead to an unwanted scatter of the data points and hide the real cluster patterns of the data. To better understand this problem, take a look at the simple example in Figure 6.1(b). The 3-dimensional dataset contains a cluster that is easily recognizable in the 2-dimensional projections as black bars. However, the noise attribute conceals the clear cluster structure in 3 dimensions and the effect becomes stronger the more noise dimensions are considered.

To cope with the described problems, different approaches including *projected* and *subspace* clustering algorithms have been investigated in the last years. These techniques assume that only a subset of dimensions is relevant for a cluster and the remaining dimensions are noise. Subspace clustering approaches further allow points to be in multiple clusters at the same time, while projected clustering approaches result in an unique assignment of points to clusters. The task of subspace and projected clustering algorithms is to identify the clusters and their subspaces. An interested reader will find a survey in [KKZ09].

Due to the exponential number of subspaces, the size of the results of subspace clustering approaches can become very large if meaningful clusterings are found in many subspaces. In contrast to it, projected clustering



(a) Mean and variances of Euclidean norm with growing number of dimensions for constant values a and b.



(b) Noisy dimension conceals the cluster that is clearly recognizable in the x-y and y-z projections (black bars).

Figure 6.1: Curse of dimensionality and problems with irrelevant dimensions in high-dimensional spaces.

approaches provide a single clustering that is usually smaller than the result of subspace clustering approaches. This property of projected clustering approaches is an advantage for large datasets and therefore in this work, we focus on projected clustering techniques.

Both, projected and subspace clustering are sophisticated data analysis techniques for high dimensional spaces but they are often connected with high computational and I/O costs, since the search in all subspaces has an exponential time complexity in the number of dimensions and often requires large number of iterations over all data. This reason hinders the development of efficient algorithms for large datasets. A broad analysis of the existing algorithms, however, showed that the P3C algorithm [MSE06] possesses a parallelization-friendly structure and is well suitable for processing large data volumes. Therefore, in this work we present P3C$^+$-MR, the first exact solution for MapReduce-based projected clustering. To the best of our knowledge, at this point of time, there are no other exact scalable implementations for projected clustering techniques and our approach is the first solution of this kind. We will also point out different problems, which occur when apply-

ing P3C$^+$-MR to huge datasets and present our solution - the P3C$^+$-MR-Light algorithm.

The main contributions of our work are:

1. Adaptation of the P3C clustering model for large datasets that results in our novel P3C$^+$ algorithm
2. Two efficient MapReduce-based implementations P3C$^+$-MR and P3C$^+$-MR-Light of P3C$^+$
3. Evaluation of proposed solution and comparison to an existing approximate sampling-based solution [CJT$^+$11]

Following sections are organized as follows: in Section 6.2 we give a short review of some well-known projected clustering algorithms, and explain our decision to choose P3C. In the following Sections 6.3 and 6.4, we describe the P3C algorithm in details, explain the statistical and algorithmic issues that make P3C not effective for large datasets and propose the necessary changes of the clustering model to deal with these problems that results in our P3C$^+$ algorithm. In Sections 6.5 and 6.6, we present P3C$^+$-MR and P3C$^+$-MR-Light, the MapReduce implementations of the P3C$^+$ and provide experimental evaluation in Section 6.7. Finally, Section 6.8 concludes this chapter.

6.2 Related Work

In the past decade, a large number of projected clustering algorithms were proposed. Similar to full space clustering approaches these techniques are based on different assumption about the data that lead to different clustering models.

PROCLUS [APW$^+$99] algorithm is a k-medoid-like approach that searches for such subspaces of clusters, which minimize the dimension-wise standard deviations of distances of points in the neighborhood of the medoids. Similar to k-medoid, the algorithm starts with a set of k medoids and iteratively improves the result by exchanging low-quality medoids by better ones as long

as the quality of the clustering is improving.
DOC [PJAM02] is a density-based approach that defines a cluster as a set of dense points in a hyperrectangle. Intuitively, an optimal projected cluster in DOC is a cluster with as many objects as possible and as large dimensionality as possible. With this goal in mind, DOC performs Monte Carlo simulations for calculating an approximate solution that is shown to be a 2-approximate solution. While k-means-based approaches are relatively easily expressible as a MapReduce program, density-based techniques that consider the object neighborhood are in general much more complicated. They require similarity join techniques as MR-SimJoin [SRT12] or MR-DSJ (cf. Section 3.3) with high data replication ratios and high communication.
Approaches including STATPC [MS08] or P3C [MSE06] belong to a class of clustering techniques that rely on statistical interpretation of the clusters. As well in P3C as in STATPC, a cluster is a set of points and dimensions in a hyperrectangle that have a significantly larger support (number of objects) than a comparable hyperrectangle where the points follow the uniform distribution. Both approaches employ algorithms that provide an approximate solution of the desired clustering.
Another class of approaches is generative models. Those, usually parametric methods, consider a clustering to be generated by a stochastic process and the main task is to learn the parameters of the underlying distributions. Learning procedures for non-trivial models as variational Bayes or other optimization techniques often require a large number of iterations over all data objects and therefore produce a high I/O workload.

An important practical issue for clustering algorithms is the number of parameters and how intuitive those are. The PROCLUS algorithm requires two parameters k - the number of clusters and l - the average number of dimensions. The DOC algorithm relies on two user-defined parameters α and β that describe the relative proportions of objects in a cluster C in order to define C as optimal. Both P3C and STATPC rely on statistical tests and require one or three confidence levels - a Poisson threshold or α_0, α_K, α_H, respectively. In DOC, P3C and STATPC the number of clusters is determined automatically based on the provided parameter values.

Despite the significant growth of data volumes and data dimensionality, there are almost no projected or subspace clustering algorithms that are able to deal with large amounts of data.
pMAFIA [NGC00] is a parallel version of the MAFIA subspace clustering algorithm, which, however, is not applicable for projected clustering. To the best of our knowledge, BoW [CJT+11] is the only work that addresses the problem of efficient projected clustering for large datasets in MapReduce. BoW is a general framework for parallelizing clustering algorithms, whose result sets are defined by means of hyperrectangles. The idea of the algorithm is to distribute the calculations to multiple machines by splitting the data into small subsets and executing the desired algorithm on these data blocks. In the end phase, the partial results of all subsets are combined by merging intersecting hyperrectangles to larger hyperrectangles. The authors propose different strategies for sampling and for the calculation part, which can either reduce the number of computations or reduce the I/O overhead. However, in both cases, BoW remains an approximate algorithm and as we will show in the experimental section, the quality of the algorithm may become very low.

The reason for our choice to use the P3C algorithm is the sound statistical model, a suitable algorithmic structure that allows for an efficient MapReduce-based solution, good quality shown in the evaluation of different projected and subspace clustering algorithms [GFM+11], and as stated in the original work [MSE06] the stability of the algorithm with respect to the chosen parameter setting.

6.3 Original P3C algorithm

In this section, we provide necessary definitions and describe the original P3C algorithm. Then, in the following Section 6.4, we will identify the problems of this solution in the large data scenario and present our novel P3C+ algorithm.

6.3.1 Definitions and notation

For the sake of self-containment, we introduce notation and definitions, which are based on the original P3C definitions. Moreover, we summarize all used symbols of this chapter in Table 6.1. Let $D = \{x_{ij} | i \in [1, n], j \in [1, d]\}$ be a d-dimensional set of points of cardinality n and $A = \{a_1, \ldots, a_d\}$ be a set of attributes of the dataset D. Similar to original work, we assume without loss of generality each attribute to be normalized on the [0,1] range.

Definition 6.1 (Interval, support, support set) *An* interval *on the attribute a_j in the range $i_l \leq x \leq i_u$ is denoted by $I_{a_j} = [i_l, i_u]$ and the width of this interval by $width(I) = i_u - i_l$. The* support set *of I, denoted by $SuppSet(I_{a_j})$, represents the set of objects x_{ij} located in the interval I_{a_j} in dimension a_j, i.e., $\{x_{ij} \in D | i_l \leq x_{ij} \leq i_u \wedge j = a_j\}$. The* support *$Supp(I_{a_j})$ of I_{a_j} is the cardinality of its support set $SuppSet(I_{a_j})$.*

Definition 6.2 (p-signature) *A* p-signature *$\boldsymbol{S} = \{I_{a_{i_1}}, \ldots, I_{a_{i_p}}\}$ is a set of p intervals with disjunct attributes $a_{i_j} \in [1, d]$. The* support set *of $\boldsymbol{S}$ is the intersection of the support sets of the containing intervals, i.e., $SuppSet(\boldsymbol{S}) = \{x \in D | x \in \bigcap_{I \in \boldsymbol{S}} I\}$. The support of $\boldsymbol{S}$ is the cardinality of $SuppSet(\boldsymbol{S})$. The set of attributes of a p-signatures $\boldsymbol{S}$ is denoted by $Attr(\boldsymbol{S})$.*

Definition 6.3 (Projected cluster) *A* projected cluster *is a tuple $C = (X_i, Y_i)$ consisting of a set of points X_i and a set of* relevant *attributes Y_i. An attribute is called relevant if it follows a non-uniform distribution.*

Definition 6.4 (true p-signature) *A* true p-signature *U of a projected cluster $C = (X_i, Y_i)$ is a p-signature $\boldsymbol{S}$ that consists of smallest intervals $I_{a_{j_k}}, j_k \in Y_i$ which contain all points X_i in the attributes Y_i.*

With these definitions, we proceed with description of the P3C algorithm.

6.3.2 P3C algorithm description

In this section, we briefly introduce the clustering model and the basic algorithmic design of the P3C algorithm.

D	data base	a	an attribute
d	data dimensionality	n	database size
I	interval	$\hat{I}$	set of intervals
S	p-signature	$\hat{\mathbf{S}}$	set of p-signatures
K	cluster core	$\hat{K}$	set of cluster cores
U	true p-signature	$\hat{\mathcal{U}}$	set of true p-signatures
C	hidden cluster	$\hat{\mathcal{C}}$	set of hidden clusters
Cl	found cluster	$\hat{\mathbf{Cl}}$	set of found clusters

Table 6.1: Used notation in this chapter

6.3.2.1 Clustering model

Let $\hat{\mathcal{C}} = \{C_1, \ldots, C_k\}$ be the set of hidden projected clusters in the dataset D. The desired clustering result is the set of true signatures $\hat{\mathcal{U}}$ with:

$$\forall_{C \in \hat{\mathcal{C}}} \exists!_{U \in \hat{\mathcal{U}}} : U \text{ is the true signature of C}$$
$$\wedge \forall_{U \in \hat{\mathcal{U}}} \exists!_{C \in \hat{\mathcal{C}}} : U \text{ is the true signature of C}$$

where $\exists!$ stands for 'exists exactly one'.
The P3C algorithm approximates $\hat{\mathcal{U}}$ by generating and refining a set of so-called *cluster cores* (Definition 6.5).

6.3.2.2 Algorithmic design of P3C algorithm

Given the clustering model, we proceed with the algorithmic design of the P3C algorithm and describe how the desired solution is computed. Figure 6.2 depicts the workflow of the algorithm. The hourglasses show the time consuming steps, which are primary addressed in the P3C$^+$-MR algorithm and for which MapReduce-based implemenatitions will be provided.
In order to determine the set of true signatures $\hat{\mathcal{U}}$, the P3C algorithm employs a filter and refinement strategy. It first computes a set of cluster cores $\hat{K}$ and refines them in the second clustering step. To determine $\hat{K}$, each attribute is discretized (*histogram building* step, Step 1 in Figure 6.2) and on each non-uniformly distributed attribute, all relevant intervals ($\hat{I}$) are extracted (Step 2 in Figure 6.2). An interval counts for being relevant, if it has a significantly

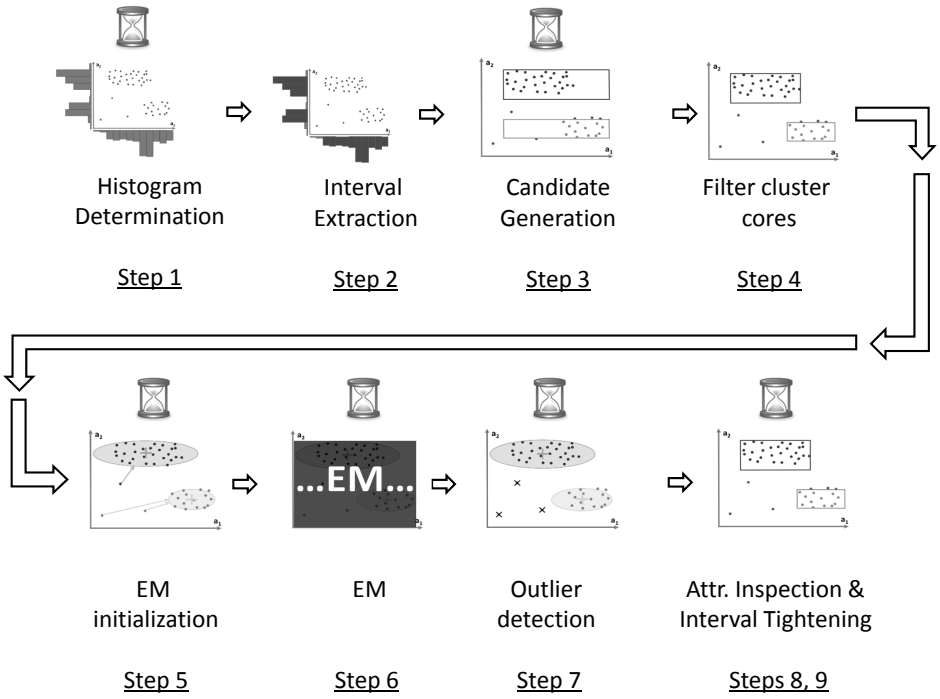


Figure 6.2: Workflow of the P3C algorithm.

high support. To select the relevant attributes, the P3C algorithm first applies the standard χ^2 test and determines the non-uniform attributes. Then, the bin with highest support is marked as relevant and removed from the histogram. The procedure is repeated as long as the remaining not marked bins are not uniformly distributed. That way, the set $\hat{I}$ of all potentially interesting intervals is computed.

After merging adjacent marked bins, the generation of *cluster cores* begins (Step 3 in Figure 6.2). For that, the intervals of all attributes are iteratively combined to high dimensional p-signatures. In each iteration, every created p-signature $\mathbf{S}_i$ is extended by an interval of an additional attribute such that its dimensionality grows by one. In order to decide whether the resulting (p+1)-signature $\mathbf{S}_{i+1}$ is a projection of a hidden cluster, it is checked whether the support of $\mathbf{S}_{i+1}$ is significantly larger than the expected support (Step 4 in Figure 6.2), formally

$$\forall_{I \in \mathbf{S}_{i+1}=\{I_1,..,I_{i+1}\}} Supp_{exp}(\mathbf{S}_{i+1} \setminus \{I\}, I) <_p Supp(\mathbf{S}_{i+1}) \tag{6.1}$$

where $x <_p y$ means that y is significantly larger than x according to the Poisson test. For this task, P3C employs the standard Poisson statistics. Only p-signatures that satisfy this test are extended by further attributes in the following iterations.
The expected support of a Signature $\mathbf{S} \cup \{I_a\}$ is calculated under the assumption that the support set of $\mathbf{S}$ is uniformly distributed on the attribute a as

$$Supp_{exp}(\mathbf{S}, I_a) = Supp(\mathbf{S}) \cdot width(I_a). \tag{6.2}$$

Please note that by assumption, $width(I_a)$ is always smaller or equal to 1, such that the expected support either remains equal or decreases with growing number of attributes.

Definition 6.5 (Cluster core) *Let K be a p-signature. K is a* cluster core *iff it contains (1.) only and (2.) all relevant intervals out of $\hat{I}$ that represent a hidden cluster, s.t.:*

1. *For any q-signature $\mathbf{Q} \subset K$, q in $\{1, .., p-1\}$ and any interval $I \in K \setminus \mathbf{Q}$, it holds that:*
$$Supp(\mathbf{Q} \cup \{I\}) >_p Supp_{exp}(\mathbf{Q}, I)$$
2. *For any interval $I \in \hat{I} \setminus K$, it holds that*
$$Supp(K \cup \{I\}) \not>_p Supp_{exp}(K, I)$$

The computed cluster cores $\hat{K}$ (i.e. the subset of generated p-signatures that apply to Definition 6.5) are considered to be approximations of projections of the real clusters $\hat{\mathcal{C}}$. To refine the cluster cores the *expectation maximization (EM) algorithm* is employed (Step 6 in Figure 6.2). The initial mean and covariance matrix parameters of the Gaussian components are determined from the calculated cluster cores (Step 5 in Figure 6.2). For each cluster core K_i, a Gaussian G_i is added to the initialization of the EM. By considering only dimensions that are relevant to at least one cluster core, the EM algorithm is

executed in the lower dimensional subspace A_{rel}:

$$A_{rel} = \{a \in A | \exists_{K \in \hat{K}} : a \text{ is relevant for } K\} \tag{6.3}$$

Given a set of k cluster cores, the resulting Gaussian components $\tilde{G}_i$ of the EM algorithm are converted into a set of k projected clusters $\hat{\mathbf{Cl}} = \{\mathbf{Cl}_1, .., \mathbf{Cl}_k\}$:

$$\begin{aligned} &\mathbf{Cl}_i = (X_i, Y_i), \text{ with} \\ &\quad x \in X_i \iff i = argmax_{i \in \{1..k\}}(p(x|\tilde{G}_i)) \\ &\quad a \in Y_i \iff a \text{ relevant for } K_i \end{aligned}$$

Since the EM algorithm assigns all points, both true members and outliers, to the clusters, outlier detection has to be applied to remove the outliers (Step 7 in Figure 6.2). The P3C algorithm applies standard multivariate outlier detection techniques [RZ90] (outlier detection step). The employed technique first computes for each member x of cluster $\mathbf{Cl}_i$ the Mahalanobis distance d_{Mah} in A_{rel} based on the mean and covariance matrix of G_i. The points for which d_{Mah} is larger than the critical value of the χ^2 distribution with $|A_{rel}|$ degrees of freedom at a confidence level of $\alpha = 0.001$ are then considered as outliers.

An additional *attribute inspection* step is executed to find relevant attributes of clusters $\mathbf{Cl}_i$, which were missed in the cluster core generation step. Similar to the relevant interval detection step in the beginning, the additional relevant attributes for cluster $\mathbf{Cl}_i$ are determined by building a histogram for the members of $\mathbf{Cl}_i$ and finding attributes A_i that are not uniformly distributed (Step 8 in Figure 6.2). The correct intervals of the output signatures are determined in the last *interval tightening* step (Step 9 in Figure 6.2). For each projected cluster $\mathbf{Cl}_i$, a signature $\mathbf{S}_i^{output}$ is provided with:

$$\begin{aligned} \mathbf{S}_i^{output} =& \{I_a = (i_{l,a} i_{u,a}) | a \in A_i\} \\ & \wedge i_{l,a} = min_{x \in \mathbf{Cl}_i}(x_a) \wedge i_{u,a} = max_{x \in \mathbf{Cl}_i}(x_a) \end{aligned}$$

The final result of the P3C algorithm is the set of output signatures $\hat{\mathbf{S}}^{output} = \{\mathbf{S}_i^{output} | K_i \in \hat{K}\}$.

6.4 Boosting P3C model for Big Data

In this section, we discuss the reason why the original P3C algorithm is not applicable in the large scale scenario. We will reason why the employed statistical significance test is not sufficient when dealing with huge datasets and propose a statistically well-founded extension. We will also introduce several algorithmic improvements that considerably increase the quality of the clustering results. We start with a discussion of statistical issues in Section 6.4.1 and continue with algorithmic improvements in Section 6.4.2.

6.4.1 Statistical issues when dealing with Big Data

The P3C algorithm makes use of different statistical techniques for determining the overall clustering result. In this section, we discuss the statistical issues of Sturge's rule and the Poisson test in the cluster generation step for large datasets and introduce our solutions.

6.4.1.1 Sturge's Rule

To determine the optimal number of bins in every dimension the P3C algorithm uses the Sturge's rule

$$\text{number bins} = \lceil 1 + \log_2 n \rceil .$$

However, as was shown [Sco92], Sturge's rule tends to oversmooth the histograms, i.e., it proposes a too small number of bins for a good approximation of the underlying distribution. Oversmoothed histograms can disturb the identification of relevant bins in the histogram building processing, and at the same time make the overall clustering result more imprecise. Additionally, as stated in [Hyn95] Sturge's rule is not well-founded and produces

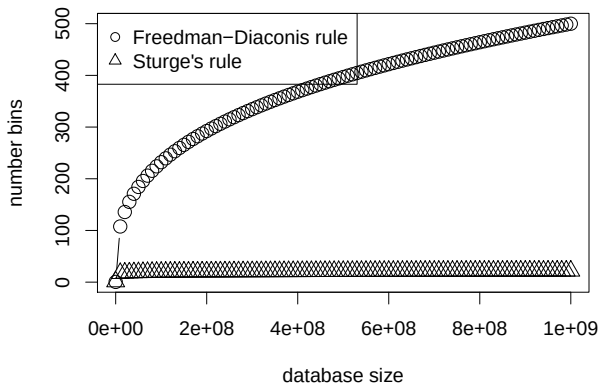


Figure 6.3: Number of bins estimated by Freedman-Diaconis and Sturge's rules

similar results to well-founded approaches for small sample sizes only. Alternative heuristics including the Freedman-Diaconis rule provides much more reliable results and should be used instead. According to the Freedman-Diaconis rule, the bin size is determined by:

$$\textit{bin size} = 2IQR(x)n^{-\frac{1}{3}},$$

where n is the sample size and $IQR(x)$ is the interquartile range of the data x. Since the determination of the precise IQR value is a data and computationally intensive task, we assume in our work that every dimension follows the uniform data distribution such that $IQR(x) = \frac{1}{2}$. As Figure 6.3 shows, despite this simplification, the utilization of Freedman-Diaconis rule leads to much more accurate approximations (due to the larger number of bins of the histogram) of the given data distributions for large datasets and therefore to more exact clustering results.

6.4.1.2 Poisson test in cluster core generation step

The second issue that may lead to much more problematic cases is the employed hypothesis test for the determination of significantly large supports of constructed p-signatures in the cluster core generation step. By means of

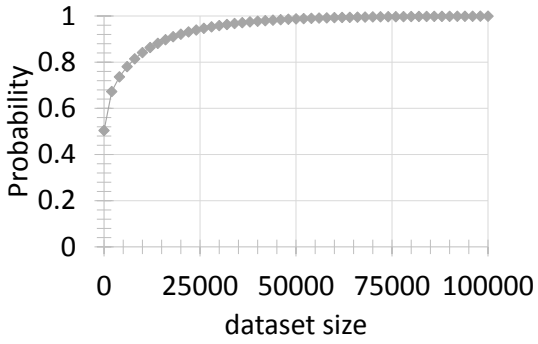


Figure 6.4: Probability to observe $101\% \cdot \mu$ objects in a hyperrectangle for growing average size. For sufficiently large datasets each the probability is almost 100%.

a Poisson distribution, the actual supports of p-signatures are tested against the respective expected supports. The result of the computation is a p-value - a probability to observe the support as extreme as or more extreme than the actually observed support, if the test statistic really were distributed as it would be under the null hypothesis (i.e., that the data is uniform distributed). However, this probability strongly depends on the database size under consideration. The p-values will monotonously decrease as the dataset size grows with constant relative deviation in bin supports. To see this, remember that the Poisson distribution can be approximated by a Gaussian with $\mu = \lambda$ and $\sigma = \sqrt{\lambda}$. I.e., the standard deviation of the normal distribution grows with a square root of the average number of elements per bin. For a linearly growing average number of elements per bin, this means that the standard deviation grows much slower than the constant relative deviation and the Poisson test will consider even relatively small deviations from the average as significant as long as the data size is large enough. The results of simulations reported in Figure 6.4 support this theoretical result. For growing average number of elements μ the probability to observe at least $101\% \cdot \mu$ elements is depicted. As one can see, for sufficiently large datasets the probability becomes almost 100%. In this example, $101\% \cdot \mu$ is an arbitrary chosen value. However, one will observe the same curve for every other value if a sufficiently large datasets is employed.

From the statistical hypothesis point of view, this result is expectable and

desired. A growing number of points in the sample increases the power of the test, i.e, it increases the probability of the test to reject the null hypothesis, if the alternative hypothesis is true.
The problematic part in our context is that the Poisson test only determines the significance of a deviation but not the *strength of the effect* (A relative deviation of one percent might be significant, but not relevant for our cause). Therefore, the P3C algorithm will consider p-signatures with a relatively small support as relevant for further computation. This, on its hand, results in a very large number of identified candidate clusters, and in the end in a low quality of the clustering. We will show the described effects in the experimental Section 6.7.

To tackle this problem, we extend the cluster core generation step and employ a complementing class of tests, called *effect size tests* [Ric96]. An effect size is a measure of the strength of a phenomenon that complements significance tests by providing the information how *large* an effect is. In the considered problem, we define the strength of a phenomenon as the relative deviation of the actual support $Supp(\mathbf{S} \cup \{I_a\} = \mathbf{S}_{\cup I})$ of the p-signature $\mathbf{S}$ and interval I_a under consideration of the expected support (cf. Equation 6.2). The definition of the desired effect size corresponds to the *Cohen's d* statistics:

$$\text{Cohen's d}_{cc} = \frac{Supp(\mathbf{S}_{\cup I}) - Supp_{exp}(\mathbf{S}_{\cup I})}{\sigma}, \tag{6.4}$$

where σ is the variance of expected number of objects such that the Cohen's d_{cc} calculates the 'amount of variation' from $Supp(\mathbf{S}_{\cup I})$ to $Supp_{exp}(\mathbf{S}_{\cup I})$ ('cc' stands for cluster core). By defining the variance parameter $\sigma = Supp_{exp}(\mathbf{S}_{\cup I})$, the resulting value is the desired relative deviation.
We introduce a new parameter $\theta_{cc} > 0$ that controls the desired strength of the effect. Consequently, a p-signature is only interesting, if $Supp(\mathbf{S}_{\cup I})$ is significantly larger than $Supp_{exp}(\mathbf{S}_{\cup I})$ and the effect size is larger than or equal to the specified threshold: $\theta_{cc} \leq$ Cohen's d_{cc}.

In fact, the χ^2 possess similar properties as the Poisson test. For larger datasets, the power of the test grows and the bins with a relative small deviation from the average bin support (small effect) will nevertheless be

considered as relevant. Assuming a uniform distributed noise (as the P3C clustering model does), this, however, should not influence the clustering result in a negative way. For datasets, in which the above assumption does not hold and the noise shows a non-uniform distribution, a possible solution to tackle the problem is the employment of the complementary effect size tests.

6.4.2 Improvement of the algorithmic design

In this section, we introduce the algorithmic changes of the original P3C algorithm that result in our novel P3C$^+$ algorithm. These algorithmic changes aim at improving the overall clustering quality and are motivated by problems that we discovered during our experimental evaluation of the original P3C algorithm on large datasets.

6.4.2.1 Cluster Core Redundancy

We extend Definition 6.5 of cluster cores by a *redundancy constraint*. A signature is *redundant*, if it describes only an intersection of hidden clusters $\hat{\mathcal{C}}_{sub} \subseteq \hat{\mathcal{C}}$ in a subspace of the united relevant intervals of $\hat{\mathcal{C}}_{sub}$. Such a redundant signature provides misleading information (presence of a cluster that does not exist) and should thus be removed from the set of cluster cores.

Figure 6.5* illustrates an example of a redundant signature. C_1 and C_2 consist of 50 data points each, where C_1 is clustered in the $\{a_1, a_3\}$ subspace and C_2 in $\{a_1, a_2\}$. Given $\hat{I} = \{I_1, .., I_4\}$ and $width(I_i) = 0.1, i \in \{1..4\}$, the cluster core generation step finds the 3 depicted 2-signatures. Obviously, $\mathbf{S}_1$ and $\mathbf{S}_2$ show enough support to pass the Poisson test with

$$Supp(\mathbf{S}_i) = 50 >_p 1 = 100 \cdot 0.1 \cdot 0.1 = Supp_{exp}(\mathbf{S}_i), i \in \{1, 2\},$$

Assuming that both clusters are uniformly distributed in their respective irrelevant dimension, the support of $\mathbf{S}_3$ becomes $50 * 0.1 + 50 * 0.1 = 10$. On a confidence level $\alpha = 10^{-6}$, it holds that $1 <_p 10$, such that $\mathbf{S}_3$ also passes the Poisson test.

*Image from [Wel13].

Obviously, signature $\mathbf{S}_3$ is redundant to $\mathbf{S}_1$ and $\mathbf{S}_2$ and should thus be removed from the set of cluster cores.

In order to identify redundant signatures, we exploit the lower ratio of $\frac{Supp}{Supp_{exp}}$ of a redundant signature, compared to the intersecting signatures it originates from. We define redundancy of a signature $\mathbf{S}$, given a set of signatures $\hat{\mathbf{S}}$ as follows:

$$\mathbf{S} \text{ redundant in } \hat{\mathbf{S}} \iff \mathbf{S} \subseteq \cup_{\mathbf{S}_i \in \hat{\mathbf{S}} \wedge \mathbf{S}_i >_r \mathbf{S}} \mathbf{S}_i, \tag{6.5}$$

where the intuitive meaning of $\mathbf{S}_1 >_r \mathbf{S}_2$ is that observing $\mathbf{S}_1$ is more interesting than $\mathbf{S}_2$:

$$\mathbf{S}_1 >_r \mathbf{S}_2 \iff \frac{Supp(\mathbf{S}_1)}{Supp_{exp}(\mathbf{S}_1)} > \frac{Supp(\mathbf{S}_2)}{Supp_{exp}(\mathbf{S}_2)}. \tag{6.6}$$

The expected support of a p-signature is calculated based on the assumption that the data is distributed uniformly on each attribute as:

$$Supp_{exp}(\mathbf{S} = \{I_1, .., I_p\}) = n \prod_{I_i \in \mathbf{S}} width(I_i). \tag{6.7}$$

When we sort the 3 signatures found depicted in Figure 6.5, we get $\mathbf{S}_3 <_r \mathbf{S}_1$ and $\mathbf{S}_3 <_r \mathbf{S}_2$. With $\mathbf{S}_3 \subseteq (\mathbf{S}_1 \cup \mathbf{S}_2)$, the redundancy criterion (cf. Equation 6.5) is met, such that the redundant signature $\mathbf{S}_3$ is successfully identified and deleted from the set of cluster cores.

6.4.2.2 Outlier Detection

The outlier detection step of the P3C algorithm removes objects from obtained clusters after the EM clustering step. Outlier detection is essential for determining tight borders of the computed clusters since every outlier objects extends the borders.

Similar to the original P3C algorithms, we employ standard multivariate outlier detection techniques [RZ90] with some modifications described later on. In order to determine if a point x of a cluster **Cl** is an outlier or not, the Mahalanobis distance of x to the cluster mean $\mu_{\mathbf{Cl}}$ is compared to a critical value d_{crit} of the χ^2 distribution (for $\alpha = 0.001$) with $|A_{rel}|$ degrees of freedom. All

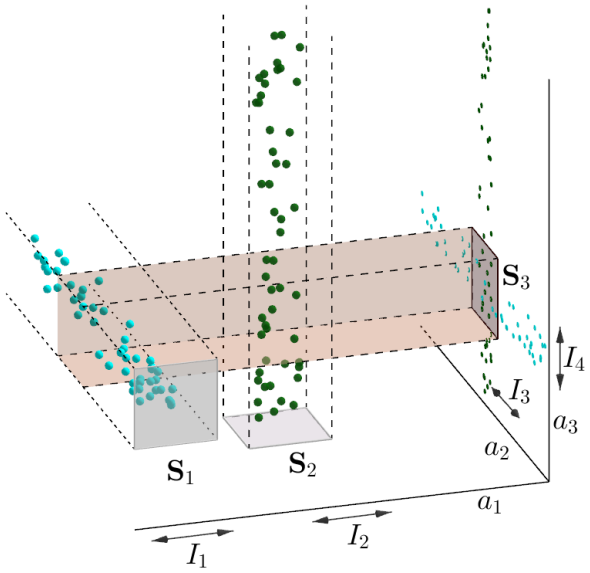


Figure 6.5: A dataset, where the intersecting region of two hidden clusters C_1 and C_2 causes an additional (redundant) signature in the $\{a_2, a_3\}$ subspace.

objects with distances above d_{crit} are marked as outliers.

The Mahalanobis distance is based on two parameters μ (mean) and Σ (covariance matrix) that are estimated from the cluster members. The *naive* approach computes both parameters from all points of a cluster **Cl**, but as stated in [RZ90] it suffers from the masking effect, i.e., the outlier objects itself influence the covariance and means and that way get masked. More robust *minimum volume ellipsoid estimator (MVE)* [RZ90] considers only points in a minimum volume ellipsoid covering half of the points of a cluster and that way improve the detection quality.
In this thesis, we investigate both approaches, the naive and the MVE based outlier detectors, on huge datasets. However, due to computational complexity of calculating the exact MVE parameter estimators, we employ an approximate MVB (minimum volume ball) solution, i.e., instead of considering half of the points of a cluster in a minimum volume ellipsoid, we compute a minimal volume ball containing half of the points. To determine the position of such a minimal volume ball, we employ a heuristic: we define the center μ_{MVB} of the MVB as dimension-wise medians of a cluster. The radius

r_{MVB} equals to the median of the distances of all cluster points to μ_{MVB}.

Given the MVB mvb with the center μ_{MVB} and the radius r_{MVB} of a cluster **Cl**, the mean and the covariance matrix of **Cl** is then determined using only points located in mvb.

6.4.2.3 Attribute Inspection

The attribute inspection is responsible for selecting relevant cluster attributes after the outlier detection step. This step corresponds to interval detection step in the original P3C with the difference that it is executed on points of a found cluster **Cl** without outlier objects and not on all points of the dataset. I.e., for the members of a cluster the histograms are generated and relevant intervals identified. The original algorithms stops at this point and accepts all found relevant intervals. This is inconsistent with the cluster core generation approach, where relevant intervals are tested using the Poisson test. Our experiments showed that the additional interval proving step improves the overall clustering result. For each additional suggested interval I_{new}, we perform the test described in Equation 6.1 Therefore, our P3C$^+$ is extended by an additional cluster support test in the cluster core detection step to which we refer as to *AI proving*.

6.5 P3C$^+$-MR in MapReduce

In this section, we provide a description of MapReduce jobs needed to implement the P3C$^+$-MR algorithm. Most parts of the algorithm are equal to a summation of statistics, which are calculated for each data object independently. This property of P3C$^+$-MR makes it fit naturally into the data parallel MapReduce framework. The desired statistic s is calculated for each object in the mapper phase and summarized in the reduce phase. We use the following shortcut notation:

$$s = \sum_{x_i} s(x_i) = \sum_{S \in Splits}^{Reduce} \sum_{x_i \in S}^{Map} s(x_i)$$

Since the calculations take about the same time for each object, perfect load balancing is achieved naturally.
Due to simplicity of the steps, we will only give a short textual description including the respective summation formula.

6.5.1 Histogram building

In this thesis, we assume normalized data space in the range $[0, 1]$. Then the computing of histograms with known number of bins m is trivial. In the mapper phase, the histograms for given data subset $Split$ are calculated and a single reducer combines the partial results to the overall histogram. The summation form for the support of bin i on dimension d is given by:

$$Supp(bin_{d,i}) = \sum_{x_i} \begin{cases} 1, & max(1, \lceil m \cdot x_{i,d} \rceil) = i \\ 0, & else \end{cases} \tag{6.8}$$

6.5.2 Relevant intervals

The determination of relevant intervals is a computationally cheap process. For a histogram with k bins and d dimensions, a simple χ^2 statistic is calculated at most $d \cdot k$ times. Even for relatively large k and d values this step is cheap in comparison to other steps, therefore its parallelization will not result in a considerable speedup of the entire process. Even more, due to the large execution overhead of the used Hadoop-based MapReduce framework, the parallel processing of this step would result in a performance degradation.

6.5.3 Cluster core generation

The cluster core generation, as depicted in Listing 6.1, involves two computationally costly steps, namely *candidate generation* (line 6) and *candidate proving* (lines 3,7). In the candidate generation step, a set of p-signatures out of given intervals and already *proved* candidates is determined. A candidate is proved if its support significantly exceeds the expected support. The

Listing 6.1: Cluster Core Generation.

Input: Intervals $I_1, .., I_n$
$Cand_1 = \{\{I_i\} | i \in \{1..n\}\}$
$Proven_1$ = *Prove Candidates*$(Cand_1)$
$k = 2$
while $Proven_{k-1}$ is not empty
  $Cand_k =$ *A Priori Candidate Generation*$(Proven_{k-1})$
  $Proven_k =$ *Prove Candidates*$(Cand_k)$
  $k{+}{+}$

$Proven = \bigcup_{i=1}^{k} Proven_i$
$Proven =$ Filter maximal Cluster Cores$(Proven)$
Output: Cluster Cores $\bigcup_{i=1}^{k} Proven_i$

counting of the support and testing for significance is called *candidate proving*.

Generating a candidate set $Cand_{p+1}$ of $(p+1)$-signatures out of a set of p-signatures is achieved by joining each two p-signatures, that have $p-1$ intervals in common. Having a set of k p-signatures, there are $c = \frac{k \cdot (k-1)}{2}$ candidate pairs for merging. Due to the quadratic complexity, the signature candidate generation consumes an unfeasible amount of time for large k, if run on a single processor. Therefore, MapReduce is employed for the parallelization of large candidate set generation using $m = \lfloor \frac{c}{T_{gen}} \rfloor$ mappers and zero reducers, for $c > 2T_{gen}$. Since each MR job adds some overhead, small candidate sets are generated in a serial manner. To approximate the size of a candidate set $Cand_{p+1}$, we use the maximal possible size of $Cand_{p+1}$ that is simply the product of the sizes of candidate sets $Cand_p^1$ and $Cand_p^2$ from which $Cand_{p+1}$ is generated.

The threshold T_{gen} should be chosen, such that the parallel version is equally fast to the serial version if $c = T_{gen}$ and depends on the available cluster. On our cluster $T_{gen} = 4 \cdot 10^7$ performed best.
The set of p-signatures is send to each mapper via the distributed cache. A set of $\frac{c}{m}$ indices is provided to each mapper, which indicates the candidate pairs to be processed. The indices range from 1 to c, each representing a pair

of p-signatures. If a pair can be merged to a $(p+1)$-signature, the mapper writes the resulting signature to the result file. The main program collects the generated $(p+1)$-signature candidates while ignoring duplicates (two pairs of p-signatures might result in the same $(p+1)$-signature).

In order to reduce I/O overhead of MR jobs, candidates are not proven at each level. If the number of generated candidates on a level j is small, the candidates on level $j+1$ are generated based on $Cand_j$ instead of $Proven_j$ (cf. line 6). When the set of collected candidates c_{sum} of several levels exceeds a threshold T_c, all candidates are proven with a single MR job. We use the following heuristic to decide where to stop the collection of candidates on level j:

$$|Cand_j| = 0 \vee (c_{sum} > T_c \wedge |Cand_j| > |Cand_{j-1}|) \Rightarrow Stop.$$

Again, the optimal setting of T_c depends on the available cluster. On our cluster $T_c = 3 \cdot 10^4$ performed best. Continuing the candidate collection as the candidate set shrinks, has proven to be most efficient throughout all of our experiments.
The multi-level candidate generation increases the total number of candidates that have to be proven as a priori pruning becomes less effective. On the other hand, it saves I/O overhead of otherwise necessary additional MR jobs. Our heuristic aims for a good trade-off of these effects.

Proving a candidate p-signature involves the determination of its actual support. Therefore, the support of each generated candidate must be calculated. This corresponds to a single MR job, where each mapper is provided all candidates $\hat{\mathbf{S}}_{all}$ and determines the set of signatures $\hat{\mathbf{S}}_{in}(x_i)$ for each data point x_i of its split, with $\hat{\mathbf{S}}_{in}(x_i) = \{\mathbf{S} \in \hat{\mathbf{S}}_{all} | x_i \in SuppSet(\mathbf{S})\}$. The reducers sums up the support counts of each cluster reported by the mappers.

To determine $\hat{\mathbf{S}}_{in}(x_i)$, each signature in $\hat{\mathbf{S}}_{all}$ is queried for containment of x_i. Since a total of 10^5 and more candidates is common, counting the support needs to be more efficient. To this end, we introduce the *Rapid Signature Support Counter* (RSSC).

The RSSC uses a bitmap representation of the problem, which is defined

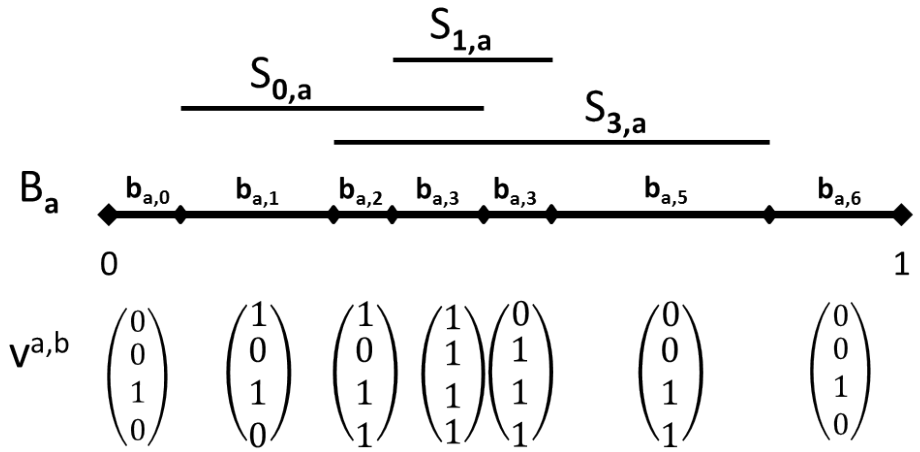


Figure 6.6: A binning B_a with bit vectors $v^{a,b}$ for each bin on attribute a with four signatures. Since a is not relevant for $\mathbf{S}_2$, the corresponding bit vector entries are set to 1

as follows. Each signature $\mathbf{S}_j$ is given a temporary $id_j \in \{0, .., |\hat{\mathbf{S}}_{all}| - 1\}$. A bit vector b of length $|\hat{\mathbf{S}}_{all}|$ represents a subset $\hat{\mathbf{S}}_{sub} \subseteq \hat{\mathbf{S}}_{all}$ with:

$$\mathbf{S}_j \in \hat{\mathbf{S}}_{sub} \iff b_{id_j} = 1$$

For each attribute $a \in A_{rel}$ (cf. Equation 6.3), we define a binning B_a. We select the bins, such that for each bin $b_{a,i}$ a bit vector $v^{a,b}$ can be given, such that for a given data point x_i the following holds

$$\forall_{\mathbf{S}_j = \{.., I_{j,a}, ..\} \in \hat{\mathbf{S}}_{all}} v^{a,b_i}_{id_j} = 0 \iff x_i \notin SuppSet(I_{j,a}),$$

where b_i denotes the bin in which x_i belongs on attribute a. Such a binning can be derived for attribute a by taking all upper and lower bounds of provided intervals on a as separators. Figure 6.6 depicts an example for such a binning with $|\hat{\mathbf{S}}_{all}| = 4$.
The desired set $\hat{\mathbf{S}}_{in}(x_i)$ is then given by the bit vector b^{in_i} with:

$$b^{in_i} = \bigwedge_{a \in A_{rel}} v^{a,b_i}$$

Having the RSSC bit masks calculated, mappers only have to bin each point in each $a \in A_{rel}$ according to B_a and aggregate the selected bit masks with the logical AND operator, both of which are very fast calculations. This way, $\hat{\mathbf{S}}_{in}(x_i)$ is calculated far more efficiently compared to the simple approach.
The bit masks are calculated by the main program beforehand and then passed to the mappers via distributed cache. Since the RSSC bit masks can be calculated with only two scans of $\hat{\mathbf{S}}_{all}$, the overhead is negligible.

Filtering cluster cores out of $Proven$ (cf. line 11 in Algorithm 6.1) is fast even for large $Proven$ sets, such that it's not necessary to parallelize this task.

6.5.4 EM

In order to initialize the EM algorithm sample means μ_C and covariances Σ_C of cluster cores have to be calculated twice. In the first iteration, μ_C and Σ_C are calculated using only the support sets of the cluster cores in order to assign outliers using the Mahalanobis distance. In the second iteration, μ_C and Σ_C are calculated using the support sets of cluster cores plus the assigned outliers.
Sample means and covariances can be calculated by two MR jobs. The first MR job calculates the linear sum l_C of the members and linear and squared sum w_C and w_{C^2} of the member weights for each cluster C:

$$l_C = \sum_{x_i} w_{C,i} x_i, \quad w_C = \sum_{x_i} w_{C,i}, \quad w_{C^2} = \sum_{x_i} w_{C,i}^2,$$

with $w_{C,i}$ being the weight of object i in cluster C. The results of the first MR job are used to calculate the sample mean $\mu_c = \frac{l_C}{w_C}$ for each cluster C and passed to the second MR job, which calculates the sample covariances σ_C of cluster C as:

$$\Sigma_C = \frac{w_C}{w_C^2 - w_{C^2}} \sum_{x_i} w_{C,i}(x_i - \mu_C)(x_i - \mu_C)^T$$

The statistics Σ_C and μ_C, which include the outliers serve as the initial Gaussian mixture model for EM. Each EM step needs another two MR jobs

[CKL+06], where the first job calculates new estimates $\tilde{\mu}_C$ and $\tilde{w}_C$, while the second job provides new covariance estimates $\tilde{\Sigma}_C$.

6.5.5 Outlier detection

For a cluster C and a member x of C, the Mahalanobis distance between x and C is compared to the critical value of a χ^2_n distribution (cf. Section 6.4.2.2). The *naive* version uses the means and covariances of the clusters as provided by the EM algorithm. Thus, a single MR job with map phase only is necessary, which is called the *OD* job. Each mapper of the OD job computes for each point x the cluster C it belongs to according to the Gaussian mixture distribution provided by the EM algorithm and the Mahalanobis distance between x and C. Finally, x is written back to the result file augmented with an additional membership attribute, which is set to the ID of C or -1, if it is considered an outlier.

The *MVB* version requires mean $\hat{\mu}_C$ and covariance $\hat{\Sigma}_C$ of the minimum volume ball MVB_C for each cluster C as explained in Section 6.4.2.2. These statistics can be extracted within three MR jobs. The first of which calculates mean and radius of each MVB_C. The following two jobs compute $\hat{\mu}_C$ and $\hat{\Sigma}_C$ for each cluster C as in the EM initialization step, but take only the points in MVB_C into account.
Mean m_C and radius r_C for each MVB_C are approximated with a single MR job, where each mapper j calculates m_C^j and r_C^j for its split and the reducer aggregates the statistics by taking the dimension-wise median of the means and radii provided by the mappers.
We define the dimension-wise median $Mdd(X)$ of a set of d-dimensional vectors X as

$$Mdd_j(X = \{x_1, ...x_n\}) = Md(x_{1,j}, .., x_{n,j}), \forall j \in \{1, .., d\},$$

where Md is the sample median.

In order to compute m_C^j and r_C^j, mapper j caches the set of all data points X_{split} of the current split. In the clean-up phase of the mapper, X_{split} is sorted

once with respect to each dimension to calculate $m_C^j = Mdd(X_{split})$. Next, the Euclidean distance between each point in X_{split} and m_C^j is calculated and stored in an array G. The radius of MVB_C^j is then given by $r_C^j = Md(G)$. In the end, the mapper writes m_C^j and r_C^j for each cluster C to the output.
After calculating MVB' means and covariances, the OD job is run using $\hat{\mu}_C$ and $\hat{\Sigma}_C$ instead of μ_C and Σ_C.

6.5.6 Attribute Inspection

In this step, one MR job is needed to calculate a histogram for each cluster core. To this end, Equation 6.8 is slightly adjusted for this task to

$$bin_{C,d,i} = \sum_{x_i \in C} \begin{cases} 1, & max(1, \lceil mx_{i,d} \rceil) = i \\ 0, & else \end{cases}$$

with $bin_{C,d,i}$ being the ith bin on dimension d on the histogram of cluster C. When AI proving is used (cf. Section 6.4.2.3), another job has to be run, in order to compute the support for the augmented signatures. This is done exactly as in the cluster core generation step.

6.5.7 Interval Tightening

At this point, for each cluster, the support set and the set of relevant dimensions are determined. The interval bounds for the output signatures are calculated in a single MR job, where each mapper calculates the minimum (maximum) value within its split for the relevant dimensions of each cluster. The reducer aggregates the provided values by repeated extracting the minimum (maximum) for each dimension and cluster.

6.6 P3C$^+$-MR-Light algorithm

During our experimental evaluation, we made an interesting observation that resulted in the P3C$^+$-MR-Light algorithm. Due to the proposed changes

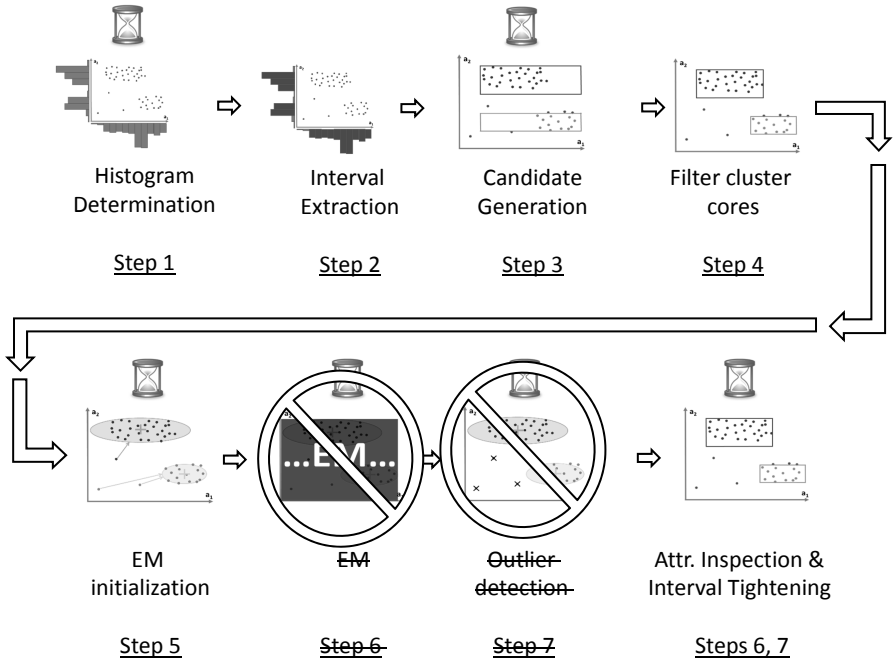


Figure 6.7: Workflow of the P3C$^+$-MR-Light algorithm. In comparison to P3C$^+$-MR, the EM-step and the outlier detection steps are omitted.

in our P3C$^+$, the generated cluster cores already provided an extremely good approximation of the hidden clusters. Moreover, the quality of the results even dropped if the following EM and outlier detection steps were executed. The main reason for this phenomenon is the fact that the outlier detection step is often not able to identify all outlier objects correctly such that the bounds of the resulting found clusters are 'blurred' in the interval tightening step.

Given the subsets of all relevant attributes A_{rel} (cf. Equation 6.3) and a cluster core K that approximates a hidden cluster C, two outlier data points x^- and x^+ that are likely to cause the blurring effect, have the following properties. Let A_C be the set of relevant attributes of C and $a_{blur} \in A_C$. The data points x^- and x^+ have to be assigned to C in EM phase, and should be close to the center μ_C of C in all attributes $a \in A^b_{rel} = A_{rel} \setminus \{a_{blur}\}$, e.g. $x_a = \mu_C, \forall a \in A^b_{rel}$. In order to blur the bounds of the interval $I_{a_{blur}}$ of K extremely, we assume $a^-_{a_{blur}} = 0$ and $a^+_{a_{blur}} = 1$. Since x^- and x^+ perfectly

match cluster C in all relevant dimensions but one, EM will output C as the most probable cluster for both points. The outlier detection algorithm will not be able to identify the outliers for the same reason. Finally, the interval tightening step will calculate the interval $I_{a_{blur}} = [0, 1]$ for K, resulting in a very poor approximation of C in attribute $I_{a_{blur}}$.
Obviously, the risk of having blurring data points rises with growing dataset size. We observed that the decreasing quality of clustering quality of P3C$^+$ with increasing dataset size (cf. Section 6.7) is directly correlated to the blurring effect.
For that reason, we also propose the P3C$^+$-MR-Light algorithm that circumvents the blurring effect by avoiding a partitioning of the dataset, as it is performed during the EM phase. P3C$^+$-MR-Light consists of all but the EM- and outlier detection steps of the P3C$^+$ algorithm and its workflow is depicted in Figure 6.7.
The histogram building in the attribute inspection step (cf. Section 6.4.2.3) requires a mapping $m(x) : D \to \hat{K} \cup \{O\}$ that maps each data point to either a single cluster or to the set of outliers O. Since P3C$^+$-MR-Light doesn't have such a partition, a mapping $m'(x) : D \to 2^{\hat{K}}$ is defined as:

$$K \in m'(x) \iff x \in SuppSet(K).$$

The histogram for a cluster core K is calculated as:

$$bin_{K,d,i} = \sum_{x_i \in SuppSet(K) \wedge |m'(x_i)|=1} \begin{cases} 1, & max(1, \lceil mx_{i,d} \rceil) = i \\ 0, & else \end{cases}$$

In order to avoid finding attributes mistakenly relevant due to the redundancy problem discussed in Section 6.4.2.1, we exclude those data points from the histograms that contribute to the support sets of more than a single cluster core.

In the following section, we proceed with evaluation of the P3C$^+$-MR, P3C$^+$-MR-Light and the BoW [CJT$^+$11] algorithms. As we stated earlier, to the best of our knowledge, BoW is the only existing framework that includes

an approximate MapReduce-based implementation of the P3C algorithm.

6.7 Experiments

In this section, we evaluate our novel P3C$^+$ approach and compare it with the original P3C algorithm in Section 6.7.4. Then, we proceed with evaluation of the MapReduce based solutions P3C$^+$-MR, P3C$^+$-MR-Light and BoW on large datasets in Section 6.7.5.

6.7.1 Dataset description

For evaluation of P3C$^+$-MR and BoW algorithm we use several synthetic and real world datasets that are described in this section.

Synthetic data To generate the synthetic data we vary three parameters: (1) number of clusters with different dimensionalities, (2) dataset size and (3) percentage of noise objects. The number of clusters is equal to 3, 5 or 7. The databases consists of 10^4, 10^5, 10^6, $5 \cdot 10^6$, 10^7, $5 \cdot 10^7$ and as extreme setting 10^9 (one billion!) objects. The percentage of noise is set to 0%, 5%, 10% and 20% of the database size. All datasets have 50 dimensions. The one billion dataset is ≈ 0.2 TByte large.

Clusters are generated in a hyperrectangular shape. The interval width of a hidden cluster on a relevant attribute varies between 0.1 and 0.3. A cluster is distributed on each relevant interval following a Gaussian distribution with $\sigma = 1$. Data points are uniformly distributed on irrelevant attributes. The clusters have 2 to 10 dimensions and may overlap with others clusters on relevant attributes. In fact, each generated dataset contains at least two clusters that overlap.

Real world data For evaluation of P3C$^+$ algorithm, we make use of the available dataset 'Colon cancer' from the standard machine learning repository UCI[†]. It consists of 62 objects with 2000 dimensions and is annotated

[†] `http://archive.ics.uci.edu/ml/`

by information whether a person has cancer or not. This dataset is used for comparison of our P3C$^+$ and the original P3C algorithms only.

6.7.2 Used measures

To evaluate our solution, we employed E4SC, F1, RNIA and CE [GFM$^+$11] evaluation measures. However, in this work we only report the results of the E4SC since it is able to detect the most important differences of a subspace/projected clustering results from the ground truth including cluster merges, wrong subspaces, wrong object assignment and many more.
A further reason for this restriction is that the remaining measures showed different drawbacks. F1, as a full space clustering measure, is not able to punish clusters in wrong subspaces and often reported too good quality for in reality bad clustering results. The CE measure showed itself as too sensitive in the case of cluster splits that often punished our competitors too much. Therefore we provide only the E4SC quality measure in this work and refer the interested reader to our web page[‡] with remaining results.

6.7.3 Parameter settings

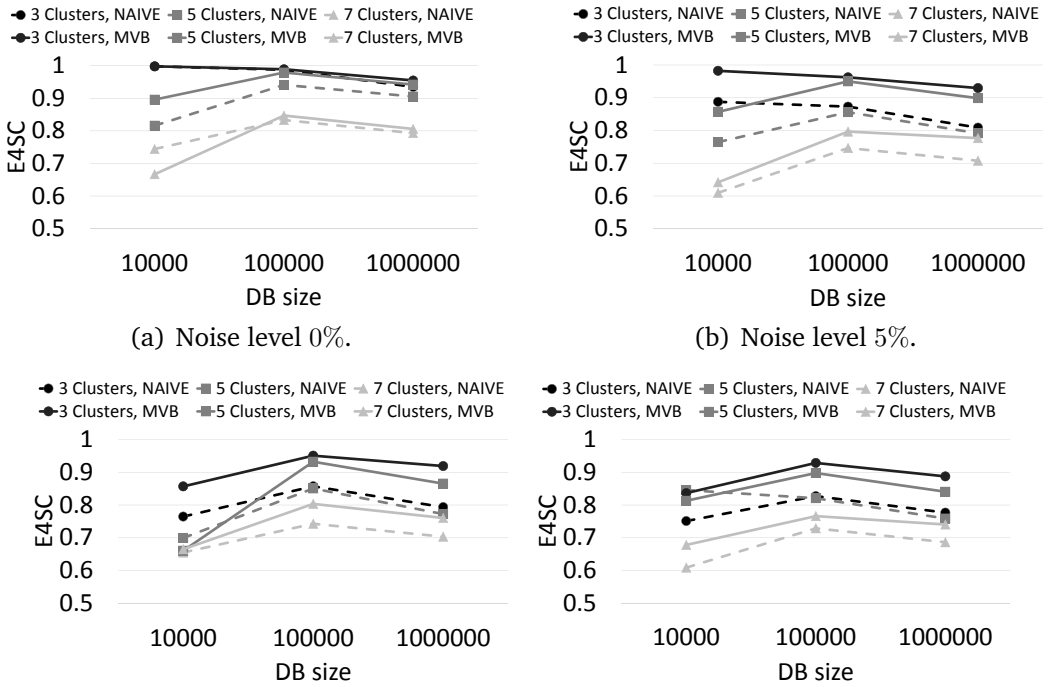
In all experiments, the parameters were set to: $\alpha_{\chi^2} = 0.001$, $\alpha_{poi} = 0.01$ and the number of samples per reducer in the BoW variant was set to 100.000.

To determine an optimal value for the θ_{cc} parameter, we executed the P3C$^+$-MR algorithm on all datasets and varied θ_{cc} in the range $[0.05, 0.5]$. In the end, the θ_{cc} was calculated as the median of all optimal values over all databases and is equal to 0.35.

6.7.4 Evaluation of the P3C$^+$ model

In this section, we evaluate our P3C$^+$ model and show how single steps improve the overall quality of the algorithm.

[‡] `http://dme.rwth-aachen.de/en/P3CMR`



(a) Noise level 0%.

(b) Noise level 5%.

(c) Noise level 10%.

(d) Noise level 20%.

Figure 6.8: Comparison of naive and MVB outlier detection steps.

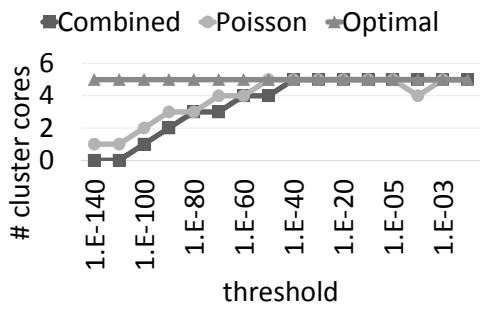
6.7.4.1 Outlier detection

In order to evaluate the quality of outlier detection techniques, we evaluated the E4SC of clustering results on databases with 10k, 100k and 1 mio objects for all noise levels (0%, 5%, 10%, 20%) and 3, 5 and 7 number of clusters. The results of the evaluation are depicted in Figure 6.8. The dotted lines represent the 'naive' outlier detection technique based on the mean and the covariance computed from all points of a cluster and the solid lines depict the results of our MVB based outlier detector. We observe that except for a single case, (noise level 10%, 5 clusters) the MVB outlier detector leads to a considerable better clustering quality, i.e., the resulting clusters much better approximate the hidden clusters. This confirms our expectation that the MVB provides a more stable estimate for the real mean and covariance of a cluster and that way leads to a better outlier detection.

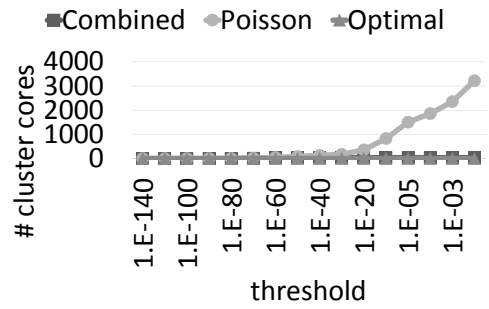
However, for the largest dataset (1 million objects) we observe that the



(a) 10k objects, 20% noise, no redundancy filtering.



(b) 10k objects, 20% noise, with redundancy filtering.



(c) 100k objects, 20% noise, no redundancy filtering.



(d) 100k objects, 20% noise, with redundancy filtering.

Figure 6.9: Effect of redundancy filtering and effect size statistics.

quality of all approaches decreases. The probable explanation is that a larger number of objects in a cluster lead to a worse estimate by the MVB and that way less outliers are identified. The exact MVE estimator will probably results in a better clustering quality but as was stated earlier the calculation of MVE is a computationally expensive step. Due to our focus on large datasets, we therefore leave this point not evaluated. However, as last remark, we observe that a sophisticated outlier detection step is crucial for good quality of the P3C and P3C$^+$ algorithms.

6.7.4.2 Redundancy filter and effect size enhancement

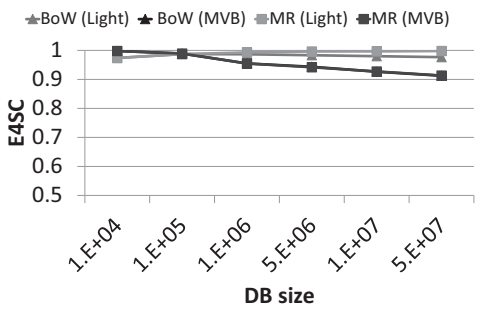
Figure 6.9 depicts the exemplary results obtained by redundancy filter described in the Section 6.9 for the synthetic datasets with 10k and 100k objects, 5 clusters and noise level of 20% percent. Each plot consists of three

curves: 'Optimal' that provides the number of hidden clusters (5), 'Poisson' that shows the behavior of the original Poisson test of the original P3C algorithm and 'Combined', that is a combination of 'Poisson' and effect size test described in Section 6.4.1.2. The Figures 6.9(a) and 6.9(c) depict the number of generated cluster cores without redundancy filtering, while Figures 6.9(b) and 6.9(d) show the effect of redundancy filter.

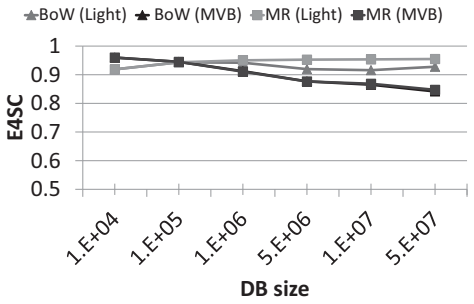
The first observation from Figures 6.9(a) and 6.9(c) is that the original 'Poisson' filter always significantly overestimate the number of cluster cores in the dataset for large threshold values. This effect is expectable since a larger threshold means that a smaller deviation of the cluster core support from the expected support is sufficient to consider the deviation as significant. Moreover, we observe that for the larger datasets (Figure 6.9(c)) the overestimation begins at a smaller threshold value ($\approx 10^{-60}$) while in the 10k dataset this effect occurs at $\approx 10^{-10}$. As explained in Section 6.4.1.2 this effect stems from the growing power of the hypothesis test. Our proposed solution to use effect size (the 'Combined' curves) considerably improves the overall result. For investigated threshold values the effect size test leads to a stagnation of the clusters core number at 40 (42) cluster for 10k (100k) datasets that is much closer to the exact number of hidden clusters (5).

The largest improvement of the results shows the redundancy filtering technique from Section 6.9 depicted in Figures 6.9(b) and 6.9(d). Already for thresholds $\geq 10^{-40}$ even the 'Poisson' test almost stabilizes at the exact number of hidden clusters and shows a deviation in two cases 10^{-4} for 10k dataset, and 10^{-2} for a 100k dataset only. The 'Combined' test shows its superiority over 'Poisson' test also in this case, as it delivers constantly correct number of cluster for all thresholds $\geq 10^{-40}$ in the 10k dataset and all investigated threshold values in the 100k dataset.

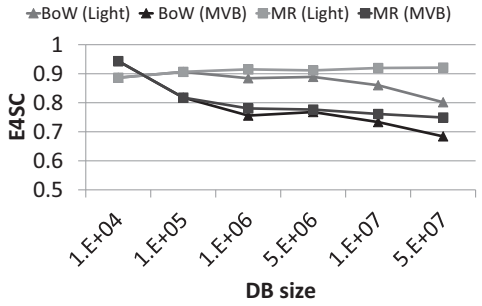
As a side remark, we want to point out that computation of cumulative probabilities with values $\geq 10^{-10}$ become infeasible due to inaccuracy of floating point arithmetic. To overcome this problem, we transform the Poisson distribution in a Gaussian distribution with appropriate parameters and determine the amount of standard deviations of the threshold. Then, we can easily determine if an observed value should be considered as significant



(a) 3 clusters, 0% noise.



(b) 5 clusters, 0% noise.



(c) 7 clusters, 0% noise.

Figure 6.10: Quality results of BoW (Light and MVB variants), P3C$^+$-MR (MR (MVB)) and P3C$^+$-MR-Light (MR (Light)) for 0% noise, different cluster sizes, database sizes.

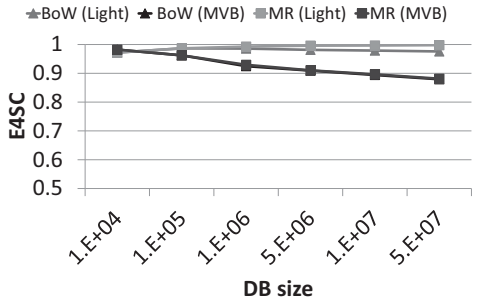
by comparison of the amount of standard deviations of the observed and expected supports.

6.7.5 Evaluation on large synthetic datasets

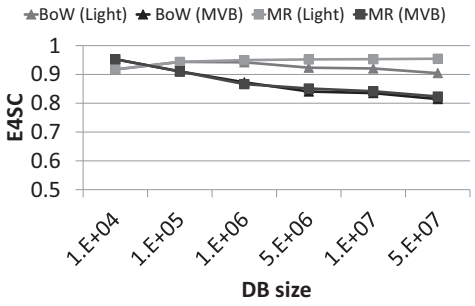
6.7.5.1 Quality results

Figures 6.10, 6.11, 6.12 and 6.13 depict the E4SC quality results of both BoW variants (Light and MVB), P3C$^+$-MR and P3C$^+$-MR-Light on synthetic datasets with up to $5 \cdot 10^7$ data points of dimensionality 50 and all examined noise levels.

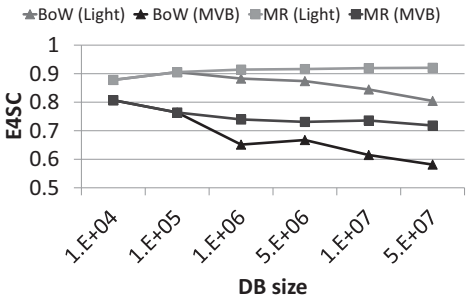
In general, we observe that the P3C$^+$-MR-Light and BoW (Light) perform better than their full equivalents with the MVB based outlier detectors. This superiority shows itself as well for datasets with growing noise levels as also



(a) 3 clusters, 5% noise.



(b) 5 clusters, 5% noise.

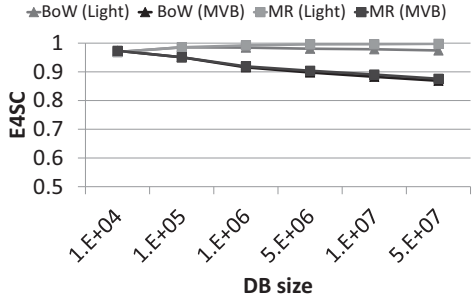


(c) 7 clusters, 5% noise.

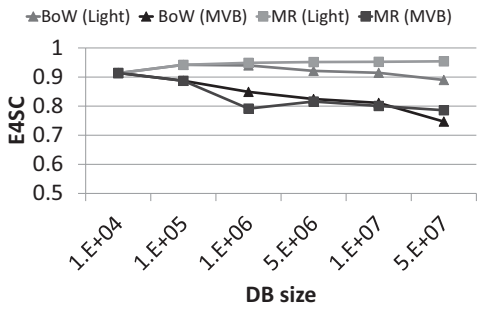
Figure 6.11: Quality results of BoW (Light and MVB variants), P3C$^+$-MR (MR (MVB)) and P3C$^+$-MR-Light (MR (Light)) for 5% noise, different cluster sizes, database sizes.

for large datasets. Furthermore, the clustering quality of the P3C$^+$-MR-Light approach even increases in most cases with growing database size, while the quality of all other approaches decreases.

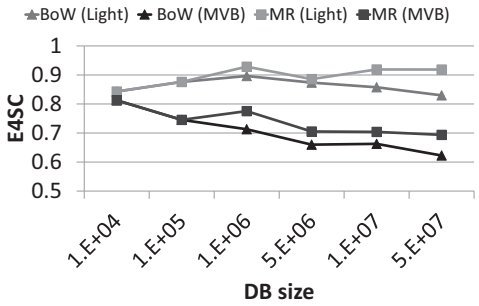
These results have several reasons. As we already mentioned earlier, the employed outlier detection techniques plays a significant role for the overall clustering result quality. Although MVB based outlier detector showed a substantial improvement in comparison to the naive outlier detector, it still not able to sufficiently good remove outlier objects. This on its hand enlarges the regions of the clusters and leads to worse quality. Due to sampling approach, both BoW variants additionally suffer from errors based on not sufficiently precisely approximated data distribution. Since the overall clustering results is computed from all subresults, even a small shift of a cluster in a single subresult will decrease the quality of the solution. With a growing dataset size,



(a) 3 clusters, 10% noise.



(b) 5 clusters, 10% noise.

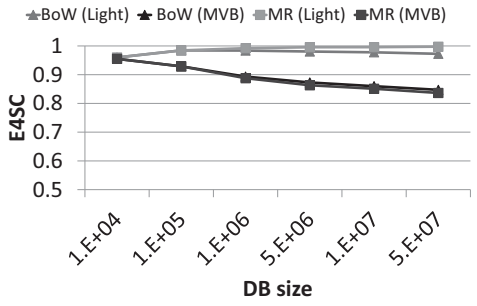


(c) 7 clusters, 10% noise.

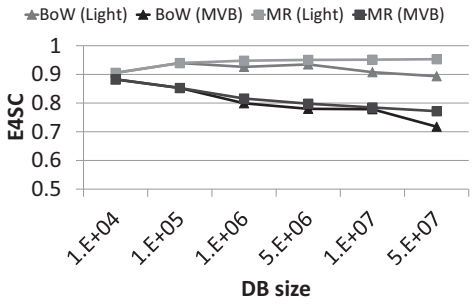
Figure 6.12: Quality results of BoW (Light and MVB variants), P3C$^+$-MR (MR (MVB)) and P3C$^+$-MR-Light (MR (Light)) for 10% noise, different cluster sizes, database sizes.

the probability that one of the samples follows another distribution than the whole dataset increases such that the above-described case occurs.

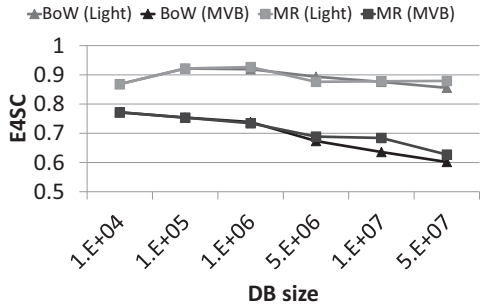
Although due to the redundancy filtering all approaches are always able to identify the right number of clusters, they show a decreasing quality with a growing number of hidden clusters in the data. This behavior arises from a combination of effects. First, as the number of clusters grows, the number of points decreases such that the identification of correct borders of the clusters become harder. Second, for a larger number of clusters the number of overlaps between clusters grows, again resulting in more 'blurred' cluster borders. Third, with growing number of clusters, the probability that a point will be associated to a wrong cluster mean during the EM clustering step or the outlier detector is not able to detect a point as outlier also grow, such that the overall clustering quality decreases.



(a) 3 clusters, 20% noise.



(b) 5 clusters, 20% noise.



(c) 7 clusters, 20% noise.

Figure 6.13: Quality results of BoW (Light and MVB variants), P3C$^+$-MR (MR (MVB)) and P3C$^+$-MR-Light (MR (Light)) for 20% noise, different cluster sizes, and database sizes.

6.7.5.2 Runtime results

In this section, we evaluate the runtimes of BoW (Light and MVB variants), P3C$^+$-MR (MR (MVB) and MR (naive)) and P3C$^+$-MR-Light (MR (Light)) algorithms on the datasets with 10^4 up to $5 \cdot 10^7$ objects (Figure 6.14). In all experiments, the number of reducers is set to 112. First, we compare the runtimes of P3C$^+$-MR with the naive and MVB based outlier detectors. A more complex computation of MVB estimator results in a $10\% - 20\%$ runtime overhead for all dataset sizes. Considering higher clustering quality achieved by the MVB-based solution (cf. Section 6.7.4.1) this overhead seems to be acceptable for real applications. Next, we examine the BoW algorithm. The obtained results are consistent with our expectations. The BoW algorithms scale linearly with the database size and with the number of used reducers. Since each reducer processes the same amount of data and the runtime

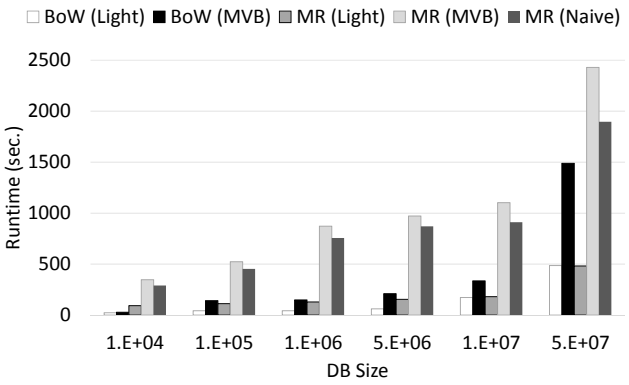


Figure 6.14: Runtime results.

of every single job is almost constant, the overall runtime equals the time needed by the map step and a time needed by reducers to process all sampled datasets. The main computational bottleneck of the BoW algorithm is therefore the number of sampled datasets. If this number is smaller than the number of available reducers, BoW will not be able to make use of the complete performance of a cluster. On the other hand, for large datasets, every reducer will have to process several data blocks. However, in general, for clusters with sufficiently many reducers, BoW provides an ideal workload distribution.

At the same time, P3C$^+$-MR and P3C$^+$-MR-Light algorithms also possess very good workload properties and in the case of P3C$^+$-MR-Light do not have reducer-dependent bottlenecks. As described in previous sections, all MapReduce jobs mainly consists of a map step and a final reduce step, performed by a single reducer. Therefore, the runtime equals the runtime of a single map step multiplied by the number of MapReduce jobs needed for clustering determination. The time of a single map step grows linearly with the database size and is not limited by the number of reducers. The main limitation is therefore the performance of the storage and network systems. In our case, the P3C$^+$-MR-Light algorithm outperforms BoW (MVB) even for small datasets and is comparably fast with BoW (Light) variant.
In comparison to other approaches, P3C$^+$-MR shows the worst runtimes. Those arise from a larger number of MapReduce jobs that have to be per-

formed and especially from the multiple iterations of the EM algorithm.

The sub-linear runtime of P3C$^+$-MR and P3C$^+$-MR-Light stems from the fact that larger inputs are processed by a larger number of mappers. Since we do not artificially split the input files in smaller chunks, the higher number of mappers resulted in an overall better runtime. For sufficiently large databases, the runtime will become linearly proportional to the size of the database.

We also executed the BoW (Light) and P3C$^+$-MR-Light on the *1 billion* dataset with 100 dimensions. On this huge dataset, the P3C$^+$-MR-Light algorithm clearly showed its superiority. While BoW (Light) needed over 9500 sec., the P3C$^+$-MR-Light produced the result in ≈ 4300 seconds.

6.7.6 Real world data

In the last experiment, we compared the quality of our P3C$^+$ algorithm to the original P3C algorithm by comparing the accuracies of the clustering results on the 'colon cancer' dataset used in [MSE06]. With 71% accuracy the P3C$^+$ outperformed the original P3C algorithm that achieved 67%.

6.8 Conclusion

In this chapter, we presented an extended version of the P3C algorithm that is capable to process huge datasets. This novel P3C$^+$ algorithm was implemented in the MapReduce framework and two versions P3C$^+$-MR and P3C$^+$-MR-Light algorithms were proposed. We compared our solution with existing approach BoW, and showed the superior accuracy of P3C$^+$-MR at costs of higher runtimes and I/O costs. In the end, we showed that an adjusted P3C$^+$-MR-Light algorithm performs best for large datasets at lower runtime and I/O costs as P3C$^+$-MR and BoW while producing highest accuracy results.

6.9 Future work: Steering the P3C$^+$ algorithm

Clustering, and as special case projected clustering techniques, are often used as supporting step for further decision making process. In the most cases, an expert evaluates and provides a semantic to the produced results. However, the P3C$^+$ algorithm depends on three parameters, which can strongly influence the size of the result set and either make the evaluation of results very time consuming or even not appropriate if too many concepts (clusters) are merged into a single cluster. For that reason, the extension of the P3C$^+$ algorithm by a steerable component is worth considering.

In the following, we briefly present a sketch how the number of clusters can be adjusted during the algorithm execution without a costly algorithm restart and discuss some open questions.

Remind the workflow of the P3C$^+$ algorithm: first, a set of one-dimensional intervals is generated. This set is used for the generation of cluster candidates (cluster cores) that are finally pruned by Poisson and effect size tests. The size of the pruned cluster core set is considered as the real number of clusters and the subsequent steps only refine the initial result. I.e., the first two steps are responsible for the size of the determined clustering.

To steer the number of found clusters, the P3C$^+$ algorithm can be started with low α_{poi} and θ_{cc} parameters that will result in a large number of cluster candidates. By storing the intermediate cluster core generation results, the usage of stricter parameter values will not require a complete restart of the algorithm. As the cluster core set for smaller (i.e., less strict) α_{poi} and θ_{cc} values contains the cluster core set of more strict values, the number of clusters can be reduced with a significantly smaller overhead by restarting the algorithm in the EM step (or if the P3C$^+$-Light is used even directly in the attribute inspection step).

Before the proposed solution can be effectively used, several questions must be answered. First, the precise effect of α_{poi} and θ_{cc} must be evaluated. Since the proposed algorithm still requires multiple phases to be performed until the final result is calculated, random choice of parameters may require many attempts to find meaningful parameter values. A heuristic that pro-

vides a guideline for the output size would significantly increase the usability of the approach.

Second question refers to the storage capacity needed to store intermediate cluster cores. Due to exponential number of subspaces, the potential number of intermediate clusters cores can become very large. In such a case, either the storage capacity can become too large, or the pruning cluster cores can become very expensive and deteriorate the performance of the steering component. This problem requires some procedure that either restrict the set of stored intermediate cluster cores to the most important ones, or provide an efficient way to deal with such a large number of subresults.

Last but now least is the question whether more phases of the algorithm can be skipped without loosing the correctness of the result. The proposed idea still requires several costly MapReduce steps and can still be too costly for a steerable approach.

Chapter 7

Discussion: Further approaches

Interestingly, despite the importance of projected and subspace clustering approaches for the analysis of high-dimensional data, almost no research on parallelization and particular on extension of these approaches for processing of large amount of data is done. From the researcher's point of view there are more or less interesting approaches measured by complexity of their parallelization. So for example, for techniques, which apply iterative-based optimization similar to k-means or EM clustering, the straightforward parallelization strategy would resemble already existing solution, in which a single iteration corresponds to a single MapReduce job. In this job, the given dataset is used for re-estimation of the previously determined parameters of the algorithm (for example, means, variances and importance of the dimensions). The iterative-based optimization paradigm is widely spread in statistically motivated approaches often described as statistical generative models. Techniques following this paradigm include for example ProClus[APW+99], LDA[BNJ03], or MVGen[GFS12] algorithms.

Despite the simple straightforward solution, the iterative optimization approaches often require dozens and depending on the used techniques or problem even hundreds and thousands of iterations. Assuming very large datasets, this results in high running times and high communication costs. A possible solution to cope with this problem is the development of approximate solutions that fast compute a good estimate for the (locally) optimal

parameter settings, which then can be refined by an exact algorithm. An example for this procedure is the canopy clustering algorithm [MNU00] that precomputes an approximate solution for the k-means or the EM algorithms.

In spite of the problem related to the large iteration number, the previously mentioned approaches are applicable to very large datasets. The formulas for the re-estimation of the parameters are usually distributive or algebraic, i.e., the new parameter value can be computed by combining partial results from data samples. An example for such a value is the mean of a dataset that can be decomposed as a sum of multiple sample means divided by the number of sample means. However, there are many approaches, which are not easily extensible for such a distributed way of computing. Even more, from the author's point of view, almost the entire class of density-based projected/subspace clustering techniques including [KKK04, AKMS07, HDG07, AKMS08] suffers from this problem. The main issue of density-based approaches is the necessity to compute the object's neighborhood. Performing this task for all objects is equivalent to computing the similarity self-join of the database as we described in the chapter related to the density-based clustering. As we showed in the experimental evaluation of our MR-DSJ and PHiDJ algorithms, the computation of the similarity self-join does not only cause a large, potentially quadratic number of distance computations but can also involve large data replication. While the first issue is an implicit property of the similarity join, the latter is closely related to the distributed memory computing architectures and particularly to MapReduce. A large data replication on its hand may exceed the storage capacities of the cluster that then prevents the computation of the results.

Besides the algorithmic problem of high data replication, many subspace clustering approaches search for clusters in an exponentially large subspace space. Although all techniques apply pruning strategies to significantly reduce the number of considered subspaces, this procedure still requires clustering of the whole dataset in all candidate subspaces. This then either results in a large number of MapReduce-jobs if each clustering in the subspaces is performed subsequently, or in a further increased computational runtime and increased communication.

Finally, to prevent redundant results, many density-based clustering approaches rely on object-wise comparison of computed clusters. Considering the earlier mentioned exponential number of subspaces, the overall amount of clusters may become overwhelming and again exceed the available resources.

For sure, not every density-based approach suffers from all the mentioned problems. However, to prevent an efficient parallelization of an algorithm in MapReduce, already a single of these problems suffices. And to the best of the knowledge of the author of this thesis, there are no approaches that addresses all these issues, such that a development of a novel parallelization-aware and data-aware model is definitely desired.

Part III

Summary and Outlook

Chapter 1 of this thesis provides background information on the Knowledge Discovery process, clustering techniques and the MapReduce framework. The parallelization of two clustering algorithms, the DBSCAN algorithm in Part I and projected clustering in Part II are the topics of the subsequent parts.

Part I Chapter 2 introduces the original DBSCAN clustering model and describes the general idea of our novel JBMR-DBSCAN approach. The JBMR-DBSCAN structure constitutes three phases: similarity self-join, core point detection and connected component search with respect to $minPts$ parameter. Both basic phases, the similarity self-join and the connected component search, are further investigated in Chapters 3 and 4, respectively. To efficiently solve the similarity self-join task for vector data, two novel approaches MR-DSJ and PHiDJ are proposed. Both utilize grid partitioning and several filtering techniques in order to reduce the number of distance computations and to reduce the communication. While the MR-DSJ approach is best suitable for low-dimensional spaces, the PHiDJ algorithm is also efficiently applicable in high-dimensional spaces. Moreover, the introduced variable grid partitioning allows the PHiDJ algorithm the handling of skewed data distributions. The experimental evaluation showed the superiority of the proposed techniques in comparison to existing solutions in the literature.

The connected component determination in undirected graphs is the second big topic of Part I. The basic idea of the proposed CC-MR algorithm is to transform the given graph into a set of star-like graphs, where each star corresponds to a single component. The result of the overall highly distributive structure of the algorithm and several optimizations including workload balancing is a very efficient approach that clearly outperforms all currently existing techniques. The CC-MR-core extension of the CC-MR algorithm further allows the determination of connected components with respect to $minPts$ parameter of the DBSCAN algorithm that further increase the performance of the overall JBMR-DBSCAN algorithm.

Finally, Chapter 5 deals with the actual JBMR-DBSCAN algorithm and its improved integrated version. Both approaches are evaluated and compared

with existing solutions from the literature. The results show that the JBMR-DBSCAN algorithm is not able to outperform its competitors on the used datasets. In spite of this, the applicability of the JBMR-DBSCAN on every metric data makes it the most general parallelized solution for the DBSCAN algorithm in the MapReduce framework existing so far.

Part II Section 6 introduces the novel MapReduce-based projected clustering algorithm P3C$^+$-MR for huge datasets. The basis of the algorithm is the P3C [MSE06] algorithm that, however, is not effectively applicable on large datasets. For that reasons, several adjustment of the P3C model were integrated and evaluated. The comparison to the existing solutions from the literature and to the original P3C algorithm shows the clear superiority of the P3C$^+$-MR algorithm in terms of cluster quality. Moreover, for very large datasets, the additionally proposed P3C$^+$-MR-Light outperforms the competitive approximate solutions in terms of running time and cluster quality.

Outlook Sections 3.9, 5.6, and 6.9 already contain possible directions of future research on similarity self-join, MapReduce-based DBSCAN algorithm and the P3C$^+$-MR algorithm. Chapter 7 additionally gives the opinion of the author of this work about chances and challenges for parallelization of further clustering approaches and proposes several directions for research.

Summarized, the MapReduce framework follows a simple parallelization model that often allows for a straightforward algorithm parallelization. However, in the last years, the scientific community investigated a tremendous number of algorithms that can be easily parallelized with this framework and proposed efficient solutions for the most of the well-known techniques. Therefore, the focus of further research should be pointed to more specialized models (as for example subspace or multi-view clustering approaches).

Part IV

Appendices

Listings

2.1 Pseudo-code of the sequential DBSCAN algorithm. 23

3.1 The MR-DSJ mapper recursively assigns objects $(id, coord)$ to neighboring partitions pid. The bitcodes 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. 47

3.2 The MR-DSJ reducer computes the join results. The bitcodes 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. . . . 48

3.3 Map function for PHiDJ 79

3.4 Reduce function for PHiDJ 83

4.1 Reducer implementation. 109

4.2 Reducer for CC-MR-core algorithm. 120

5.1 Reducer for the degree counting MapReduce job. 137

5.2 Reducer for vertex labeling. 138

5.3 Phase 1: Core/non-core object determination. 141

5.4 Modified reducer of MR-DSJ algorithm for the integrated JBMR-DBSCAN algorithm. 145

6.1 Cluster Core Generation. 184

Bibliography

[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. *Proceedings of the VLDB Endowment (PVLDB)*, 3(2):1625–1628, 2010.

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In Delis et al. [DFG99], pages 49–60.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologiesfor analytical workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):922–933, 2009.

[AC01] Domenica Arlia and Massimo Coppola. Experiments in Parallel Clustering with DBSCAN. In Rizos Sakellariou, John A. Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 326–331. Springer, 2001.

[AdBHY04] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *SIGMOD Conference*, pages 347–358. ACM, 2004.

[AKMS07] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. DUSC: Dimensionality Unbiased Subspace Clustering. In *Advances in Data Mining. Applications and Theoretical Aspects*, pages 409–414. IEEE Computer Society, 2007.

[AKMS08] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. EDSC: efficient density-based subspace clustering. In James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz, Key-Sun Choi, and Abdur Chowdhury, editors, *ACM International Conference on Information and Knowledge Management*, pages 1093–1102. ACM, 2008.

[APW+99] Charu C. Aggarwal, Cecilia Magdalena Procopiuc, Joel L. Wolf, Philip S. Yu, and Jong Soo Park. Fast Algorithms for Projected Clustering. In Delis et al. [DFG99], pages 61–72.

[ASM+12] Foto N. Afrati, Anish Das Sarma, David Menestrina, Aditya G. Parameswaran, and Jeffrey D. Ullman. Fuzzy Joins Using MapReduce. In *IEEE International Conference on Data Engineering*, pages 498–509, 2012.

[AU10] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *International Conference on Extending Database Technology*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

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

[BBKK01] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon Grid Order: An Algorithm for the

Similarity Join on Massive High-Dimensional Data. In *SIGMOD Conference*, pages 379–388, 2001.

[BFP+73] Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

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

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *SIGMOD Conference*, pages 322–331. ACM Press, 1990.

[BML10] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document Similarity Self-Join with MapReduce. In Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu, editors, *Advances in Data Mining. Applications and Theoretical Aspects*, pages 731–736. IEEE Computer Society, 2010.

[BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[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 Elmagarmid and Agrawal [EA10], pages 975–986.

[BT01] Libor Bus and Pavel Tvrdík. A Parallel Algorithm for Connected Components on Distributed Memory Machines. In *European MPI Users's Group Meeting / European MPI Users's Group Meeting*, pages 280–287, 2001.

[BUS10] Christian Beecks, Merih Seran Uysal, and Thomas Seidl. Signature Quadratic Form Distance. In Shipeng Li, Xinbo Gao, and Nicu Sebe, editors, *Proceedings of the ACM International Conference on Image and Video Retrieval*, pages 438–445. ACM, 2010.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *IEEE International Conference on Data Engineering*, page 5. IEEE Computer Society, 2006.

[CGL10] Min Chen, Xuedong Gao, and HuiFei Li. Parallel DBSCAN with Priority R-tree. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 508–511, April 2010.

[CJT+11] Robson Leonardo Ferreira Cordeiro, Caetano Traina Jr., Agma Juci Machado Traina, Julio López, U. Kang, and Christos Faloutsos. Clustering very large multi-dimensional datasets with MapReduce. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 690–698. ACM, 2011.

[CKL+06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *Advances in Neural Information Processing Systems*, pages 281–288. MIT Press, 2006.

[CLC82] Francis Y. L. Chin, John Lam, and I.-Ngo Chen. Efficient Parallel Algorithms for Some Graph Problems. *Commun. ACM*, 25(9):659–665, 1982.

[Coh09] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11(4):29–41, 2009.

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

[DFG99] Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors. *Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. ACM Press, 1999.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[DL12] Bi-Ru Dai and I-Chang Lin. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 59–66, 2012.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[DN14] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal*, 23(3):355–380, 2014.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In Li-Yan

Yuan, editor, *International Conference on Very Large Databases*, pages 27–40. Morgan Kaufmann, 1992.

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

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

[EA10] Ahmed K. Elmagarmid and Divyakant Agrawal, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010.

[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 *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[FH75] Keinosuke Fukunaga and Larry D. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, 1975.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. The KDD Process for Extracting Useful Knowledge from Volumes of Data. *Commun. ACM*, 39(11):27–34, 1996.

[GFM+11] Stephan Günnemann, Ines Färber, Emmanuel Müller, Ira Assent, and Thomas Seidl. External evaluation measures for subspace clustering. In Craig Macdonald, Iadh Ounis, and Ian Ruthven, editors, *ACM International Conference on Information and Knowledge Management*, pages 1363–1372. ACM, 2011.

[GFS12] Stephan Günnemann, Ines Färber, and Thomas Seidl. Multi-view clustering using mixture models in subspace projections. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 132–140. ACM, 2012.

[Gra13] Alexander Gras. Evaluation of MapReduce-based DBSCAN Algorithms on Big Data. Bachelor's thesis, RWTH Aachen University, 2013.

[Gre94] John Greiner. A Comparison of Parallel Algorithms for Connected Components. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 16–25, 1994.

[Haw80] Douglas M Hawkins. *Identification of outliers*, volume 11. Springer, 1980.

[HCS79] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing Connected Components on Parallel Computers. *Commun. ACM*, 22(8):461–464, 1979.

[HDG07] Jiani Hu, Weihong Deng, and Jun Guo. A Clustering Algorithm Based on Adaptive Subcluster Merging. In Ziad Kobti and Dan Wu, editors, *Canadian Conference on AI*, volume 4509 of *Lecture Notes in Computer Science*, pages 241–249. Springer, 2007.

[HGN00] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for Association Rule Mining - A General Survey and Comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[HTL+11] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. In *ICPADS*, pages 473–480. IEEE, 2011.

[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.

[Hyn95] Rob J Hyndman. The problem with sturges rule for constructing histograms, 1995.

[JKP04] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. DBDC: Density Based Distributed Clustering. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2004.

[JS08] Edwin H. Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[KKK04] Peer Kröger, Hans-Peter Kriegel, and Karin Kailing. Density-Connected Subspace Clustering for High-Dimensional Data. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *Proceedings of the SIAM International Conference on Data Mining*, pages 246–256. SIAM, 2004.

[KKZ09] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, March 2009.

[KLCY97] A. Krishnamurthy, S. Lumetta, D.E. Culler, and K. Yelick. Connected components on distributed memory machines. *DIMACS implementation challenge*, 30:1, 1997.

[KNT00] Edwin M. Knorr, Raymond T. Ng, and V. Tucakov. Distance-Based Outliers: Algorithms and Applications. *The VLDB Journal*, 8(3-4):237–253, 2000.

[Kot07] Sotiris B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. In Ilias Maglogiannis, Kostas Karpouzis, Manolis Wallace, and John Soldatos, editors, *Emerging Artificial Intelligence Applications in Computer Engineering*, volume 160 of *Frontiers in Artificial Intelligence and Applications*, pages 3–24. IOS Press, 2007.

[KSR14] Lars Kolb, Ziad Sehili, and Erhard Rahm. Iterative Computation of Connected Graph Components with MapReduce. *Datenbank-Spektrum*, 2014.

[KTF09] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In *Advances in Data Mining. Applications and Theoretical Aspects*, pages 229–238, 2009.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–94. ACM, 2011.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

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

[LV10] J.A. Lee and M. Verleysen. *Nonlinear Dimensionality Reduction*. Information Science and Statistics. Springer, 2010.

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

[MNU00] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 169–178, New York, NY, USA, 2000. ACM.

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

[MRL98] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, pages 426–435. ACM Press, 1998.

[MS08] Gabriela Moise and Jörg Sander. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In Ying Li, Bing Liu, and Sunita Sarawagi, editors, *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 533–541. ACM, 2008.

[MSE06] Gabriela Moise, Jörg Sander, and Martin Ester. P3C: A Robust Projected Clustering Algorithm. In *Advances in Data Mining. Applications and Theoretical Aspects*, pages 414–425. IEEE Computer Society, 2006.

[Mun14] J. Ian Munro. In as Few Comparisons as Possible. In Sudebkumar Prasant Pal and Kunihiko Sadakane, editors, *WALCOM*, volume 8344 of *Lecture Notes in Computer Science*, pages 5–6. Springer, 2014.

[NGC00] Harsha S. Nagesh, Sanjay Goil, and Alok N. Choudhary. A Scalable Parallel Subspace Clustering Algorithm for Massive Data Sets. In *ICPP*, pages 477–484, 2000.

[NSSH11] Zhixian Niu, Shuping Shi, Jingyu Sun, and Xiu He. A Survey of Outlier Detection Methodologies and Their Applications. In

Hepu Deng, Duoqian Miao, Jingsheng Lei, and Fu Lee Wang, editors, *AICI (1)*, volume 7002 of *Lecture Notes in Computer Science*, pages 380–387. Springer, 2011.

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 949–960. ACM, 2011.

[Pat96] Mike Paterson. Progress in Selection. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 1996.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 259–270. ACM Press, 1996.

[PJAM02] Cecilia Magdalena Procopiuc, Michael Jones, Pankaj K. Agarwal, and T. M. Murali. A Monte Carlo algorithm for fast projective clustering. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 418–427. ACM, 2002.

[PPA+12] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei keng Liao, Fredrik Manne, and Alok N. Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In Jeffrey K. Hollingsworth, editor, *SC*, page 62. IEEE/ACM, 2012.

[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 Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 165–178. ACM, 2009.

[Ric96] JohnT.E. Richardson. Measures of effect size. *Behavior Research Methods, Instruments, & Computers*, 28(1):12–22, 1996.

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

[RMCS12] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding Connected Components on Map-reduce in Logarithmic Rounds. *CoRR*, abs/1203.5387, 2012.

[RMCS13] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. *2013 IEEE 29th International Conference on Data Engineering*, 0:50–61, 2013.

[RRS00] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient Algorithms for Mining Outliers from Large Data Sets. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 427–438. ACM, 2000.

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

[RZ90] Peter J. Rousseeuw and Bert C. van Zomeren. Unmasking Multivariate Outliers and Leverage Points. *Journal of the American Statistical Association*, 85(411):pp. 633–639, 1990.

[Sco92] David W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization (Wiley Series in Probability and Statistics)*. Wiley, 1 edition, September 1992.

[SFB13] Thomas Seidl, Sergej Fries, and Brigitte Boden. MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder,

and Veit Köppen, editors, *Datenbanksysteme fÃ¼r Business, Technologie und Web (BTW)*, volume 214 of *LNI*, pages 37–56. GI, 2013.

[SG12] Neha Soni and Amit Ganatra. Categorization of Several Clustering Algorithms from Different Perspective: A Review. *International Journal*, 2(8), 2012.

[SR12] Yasin N. Silva and Jason M. Reed. Exploiting MapReduce-based similarity joins. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 693–696. ACM, 2012.

[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.

[SV82] Yossi Shiloach and Uzi Vishkin. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[Tis10] Alexander Tiskin. Parallel Selection by Regular Sampling. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2010.

[TXZ06] Yufei Tao, Xiaokui Xiao, and Shuigeng Zhou. Mining distance-based outliers from large databases in any metric space. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 394–403, 2006.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In Elmagarmid and Agrawal [EA10], pages 495–506.

[Vit85] Jeffrey Scott Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[WD10] Bin Wu and YaHong Du. Cloud-based Connected Component Algorithm. In *Artificial Intelligence and Computational Intelligence (AICI)*, volume 3, pages 122 –126, 2010.

[Wel13] Stephan Wels. Parallel Projected Clustering of Big Data with MapReduce. Master's thesis, RWTH Aachen University, 2013.

[XJK99] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Min. Knowl. Discov.*, 3(3):263–290, 1999.

[XLY08] Chuan Xiao, Wei Wang 0011, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *WWW*, pages 131–140. ACM, 2008.

[YXF11] Hui Fang Ma Yan Xiang Fu, Wei Zhong Zhao. Research on Parallel DBSCAN Algorithm Design Based on MapReduce. *Advanced Materials Research*, 301-303:1133–1138, 7 2011.

[ZHL+09] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial Queries Evaluation with MapReduce. In *GCC*, pages 287–292. IEEE Computer Society, 2009.

[ZJ03] Yanchang Zhao and Song Junde. AGRID: An Efficient Algorithm for Clustering Large High-Dimensional Datasets. In Kyu-Young Whang, Jongwoo Jeon, Kyuseok Shim, and Jaideep Srivastava, editors, *The Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 2637 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2003.

[ZLJ12] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel kNN joins for large data in MapReduce. In Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari, editors, *International Conference on Extending Database Technology*, pages 38–49. ACM, 2012.

[ZZC+00] Aoying Zhou, Shuigeng Zhou, Jing Cao, Ye Fan, and Yunfa Hu. Approaches for Scaling DBSCAN Algorithm to Large Spatial Databases. *J. Comput. Sci. Technol.*, 15(6):509–526, 2000.

Statement of Originality

This thesis would not have been possible without the close collaboration in the group of Professor Seidl. Many of the presented ideas and techniques evolved from numerous fruitful discussions in the group. While the high level of collaboration both within the group and with the students constitutes a key factor of the productive environment I found at Professor Seidls group, it makes it hard to pinpoint individual contributions. Professor Seidl is clearly to be credited with making numerous suggestions for improvements and additions to the presented work. The following provides some more detail on collaborations and support for the individual chapters.

The initial ideas of the MR-DSJ (Section 3.3) and CC-MR algorithm (Chapter 4) account to Professor Seidl; the detailed work out was done in collaboration with Brigitte Boden. The approaches were implemented by Hamza Abusalah and Roman Haag. The PHiDJ algorithm (Section 3.6) was developed by me and Professor Seidl in collaboration with Brigitte Boden; Grzegorz Stepien implemented the most parts of the code. JBMR-DBSCAN (Chapter 5) is a collaborative work with Brigitte Boden and initially with Professor Seidl; Brigitte Boden and I also advised Xueyao Jiang and Alexander Gras, who dealt with the development, partial analysis and evaluation of this algorithm in their Master's thesis and Bachelor's thesis, respectively. Finally, the P3C$^+$ algorithm (Chapter 6) is a result of the Master's thesis of Stephan Wels, which was advised by me.

List of Publications

[F09] Sergej Fries. Bestimmung des optimalen Verzweigungsgrades hierarchischer Anytime-Klassifikatoren. *Studierendenprogramm at Database Systems for Business, Technology, and Web (BTW)*, 2009.

[KGF+10] Philipp Kranen, Stephan Günnemann, Sergej Fries and Thomas Seidl. MC-Tree: Improving Bayesian Anytime Classification . In *Proc. of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 252–269 2010.

[BFF+11] Christian Böhm, Ines Färber, Sergej Fries, Ulrike Korte, Johannes Merkle, Annahita Oswald, Thomas Seidl, Bianca Wackersreuther, and Peter Wackersreuther. Efficient Database Techniques for Identification with Fuzzy Vault Templates. In *Proc. of the International Conference of Biometrics Special Interest Group, GI*, 115–126, 2011.

[BKA+11] Christoph Busch, Ulrike Korte, Sebastian Abt, Christian Böhm, Ines Färber, Sergej Fries, Johannes Merkle, Claudia Nickel, Alexander Nouak, Annahita Oswald, Thomas Seidl, Bianca Wackersreuther, Peter Wackersreuther, and Xuebing Zhou. Biometric Template Protection: Ein Bericht über das Projekt BioKeyS. *Datenschutz und Datensicherheit - DuD*, 183–191, 2011.

[BFF+11] Christian Böhm, Ines Färber, Sergej Fries, Ulrike Korte, Johannes Merkle, Annahita Oswald, Thomas Seidl, Bianca Wackersreuther and Peter Wackersreuther. Filtertechniken für geschützte biometrische Datenbanken. In *Database Systems for Business, Technology, and Web (BTW)*, 379–389, 2011.

[SBF12] Thomas Seidl, Brigitte Boden, and Sergej Fries. CC-MR - Finding Connected Components in Huge Graphs with MapReduce. In *Proc. of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 458–473 2012.

[SFB13] Thomas Seidl, Sergej Fries, and Brigitte Boden. MR-DSJ: Distance-based Self-Join for Large-scale Vector Data Analysis with MapReduce. In *Database Systems for Business, Technology, and Web (BTW)*, pages 37–56. GI, 2013.

[FBS+14] Sergej Fries, Brigitte Boden, Grzegorz Stepien and Thomas Seidl. PHiDJ: Parallel Similarity Self-Join for High-Dimensional Vector Data with MapReduce. In *Proc. IEEE 30th International Conference on Data Engineering (ICDE)*, 2014.

[FWS14] Sergej Fries, Stephan Wels and Thomas Seidl. Projected Clustering for Huge Data Sets in MapReduce. In *International Conference on Extending Database Technology (EDBT)*, 2014.