Scalable Duplicate Detection

utilizing Apache Spark

Master's Thesis

University of Hamburg
MIN Faculty
Department of Informatics

presented by:	Niklas Wilcke
email:	1wilcke@informatik.uni-hamburg.de
mat. no.:	6402007
1st assessor:	Prof. Dr.-Ing. Norbert Ritter
2nd assessor:	Dr. Dirk Bade

Abstract

English

This thesis deals with the challenge of designing, prototypically implementing and evaluating the scalable duplicate detection framework SddF. Scalable in this case means from a single desktop machine and small amounts of input data to a cluster of theoretical hundreds of nodes and great amounts of input data. Entity Resolution, Record Linkage or (Near) Duplicate Detection is the discipline of assigning every tuple of an input set to a real world entity. In the resulting mapping, more than one tuple can be assigned to the same real world entity. These groups are duplicate clusters in the notion of a fuzzy human perception.
There are approaches and projects for either single desktop computers like Febrl [CCH04] or clusters like Dedoop [Kol14], but the projects are specialized to either platform. Our approach runs completely in memory and therefore scales from a single machine to a cluster. To achieve this scalability we make use of the Hadoop MapReduce successor Apache Spark, which claims to be the first interactive cluster computing framework.
The two core contributions are the adaptation of the duplicate detection process of Christen [Chr12] to a distributed environment and the design of the duplicate detection framework. The Frameworks architecture is based on a pipelining approach, which is also used by many other related projects. Additional contributions are the linear typesafe pipelining framework called PipE and a new music input data set extracted from the Musicbrainz database containing 10^7 tuples. The data set has a size of approximately 1 GB which is small compared to data sets typically processed by Hadoop or Spark. For the duplicate detection domain this amount is large. Only one other publication evaluates its algorithms implemented on top of MapReduce using such great amounts of input data.
The evaluation focuses on time and space which are the runtime of the whole process and the amount of entities contained in the intermediate results. The size and shape of the input data and the number of cluster nodes have been varied during the evaluation to investigate their impact on runtime and space. The results show that a duplicate detection workflow can scale from a single machine to a cluster of five nodes with good runtimes on both setups. The present study aims to bring both fractions of single machine approaches and distributed ones together and builds a new open foundation for further research on duplicate detection.

Deutsch

Diese Arbeit behandelt das Design, die prototypische Implementierung und Evaluation des skalierbaren Duplikaterkennungsframeworks SddF. Skalierbarkeit bedeutet in diesem Fall von einem einzelnen Computer und geringen Datenmengen bis zu Computerclustern von mehreren hundert Rechnern und großen Datenmengen. Duplikaterkennung, oft auch als (Near) Duplicate Detection oder Record Linkage bezeichnet, ist die Problemstellung jedem Tupel einer Eingabemenge eine Realweltentität zuzuordnen. Das gesuchte Mapping ordnet einer Realweltentität beliebig viele Tupel zu. Diese Mengen von Tupeln sind die Duplikatcluster, die nach Möglichkeit einer menschlichen unscharfen Auffassung von Duplikaten entsprechen sollen.
Es gibt bereits Ansätze und Projekte für Einzelrechner wie Febrl [CCH04] oder für Computercluster wie Dedoop [Kol14]. Allerdings sind diese Projekte auf die jeweilige Hardwareplattform Einzelrechner oder Computercluster beschränkt. Der Ansatz dieser Arbeit ist es beide Plattformen zu unterstützen, von einem Einzelrechner bis zu einem Computercluster von hunderten von Knoten. Um diese Skalierbarkeit zu erreichen wird das verteilte Cluster-Computing-Framework Apache Spark benutzt, welches Datensätze vollständig im Hauptspeicher vorhält.
Die zwei hauptsächlichen Beiträge dieser Arbeit sind einerseits die Adaption des Duplikaterkennungsprozesses von Christen [Chr12] für eine verteilte Umgebung und das Design des Duplikaterkennungsframeworks. Die Architektur des besagten Frameworks basiert auf dem Pipeline-Ansatz, der auch von diversen anderen Projekten in diesem Bereich gewählt wurde. In diesem Zuge ist das Pipeliningframework PipE entstanden, welches für die Implementierung genutzt wird. Zudem wurde ein Datensatz mit 10^7 Tupeln für die Duplikaterkennung erstellt, welcher aus der freien Musicbrainz Datenbank extrahiert und anschließend künstlich mit Duplikaten versehen wurde. Der Datensatz hat eine Größe von ungefähr 1 GB, welche für Spark klein ist, für die Domäne der Duplikaterkennung ist sie allerdings groß. Nur eine andere Publikation evaluiert ihre auf Hadoop MapReduce basierende Implementierung mit Datenmengen dieser Größenordnung [VCL10].
Die Evaluation dieser Arbeit konzentriert sich auf Laufzeit und Speicherverbrauch. Während der Experimente wurden die Größe und Beschaffenheit der Eingabedaten und die Anzahl der Clusterknoten variiert, um deren Auswirkung auf die Laufzeit und den Speicherverbrauch zu untersuchen. Die Ergebnisse zeigen, dass der Duplikaterkennungsprozess und die Implementierung von einem Einzelrechner zu einem Computercluster von fünf Knoten skaliert. Diese Arbeit vereint bestehende Ansätze der Duplikaterkennung in einem neuen Framework, welches eine offene Grundlage für weitere Implementierungen und Forschung im Bereich der Duplikaterkennung bieten soll.

Acknowledgement

I want to thank my supervisors Wolfram Wingerath and Steffen Friedrich for being always ready to help. Also thanks to Fabian Panse who gave advices with his deep theoretical knowledge about duplicate detection. Special thanks goes to Kai Hildebrandt for an intensive cooperation during the writing of our master's theses and exploring the world of pasta. I want to thank the first assessor Prof. Dr.-Ing. Norbert Ritter and the second assessor Dr. Dirk Bade for the assessment and the feedback they gave to me to improve my further work. Many thanks in general goes to my better half Annika Wendt who always supported me. For his effort to improve my style of writing I want to thank my brother Timo Wilcke. Last, but not least, I want to thank my parents and family for making this graduation possible.

Contents

1. Introduction

1.1. Motivation

1.1.1. Why Duplicate Detection?

Three main use cases motivate to do duplicate detection. The first one is to increase the quality of existing databases by merging duplicates. The second one is to merge existing databases in a clean way. This one is a sub-case of the first one because the databases can be merged and afterwards "cleaned" by removing the duplicates. We consider this one to be a separate problem class because the relation between tuples of the one set is restricted to tuples of the other set. The third case is to do search result aggregation. Search result aggregation is a special view on a search result with all similar results aggregated into one single representative result element. In general, the user wants to find unique result items. Uniqueness in this case means a fuzzy and human interpretation of similarity. This technique is also important for meta search engines because they have to merge all result lists of different search engines. Google announced in 2008 that the search index has reached 10^{12} unique websites[1]. One target out of reach at the moment would be the application of duplicate detection workflows on huge search indexes of such a size. This thesis focuses on input set sizes of up to 10^7 tuples, resulting in a naive quadratic search space of 10^{14} tuple pairs. As mentioned above, duplicate detection is not only useful for large search indexes, but it can also be used to increase the data quality in large databases like Musicbrainz[2] or OpenStreetMap[3].

1.1.2. Why Modular and Open Source?

The concepts presented in this thesis and their prototypical implementations are meant to invite and encourage others to contribute new algorithms. Modularity in this case enables a contributer to share his new algorithm without knowing the whole library. The library is open source because I think any work done by a public organization, especially universities, should be with respect to privacy concerns as open as possible. Another

[1]Source: `https://googleblog.blogspot.de/2008/07/we-knew-web-was-big.html`
[2]`https://musicbrainz.org/`
[3]`https://www.openstreetmap.org`

motivation to disclose the source code is to enable others to reproduce measurements done in the evaluation Chapter 5. The source code of the SddF is released under the GPL3[4] on GitHub[5].

1.1.3. Disclaimer

This work was done in the strong believe in free software and free knowledge. Like any technique it can be acquired for inhuman purposes. Especially Big Data became a synonym for eavesdropping and mass surveillance. The author distances himself from such intentions, but also wants to clarify the usefulness of duplicate detection in a free, open and peaceful Internet.

1.2. Thesis Objective

Adapting the duplicate detection process to a distributed system.

The duplicate detection process of Christen [Chr12] seems to be suitable for parallelization, but some aspects have to be considered when using it in a distributed environment. Chapter 3 covers the whole process with respect to scalability.

Is Apache Spark suitable to implement a scalable duplicate detection framework on top of it?

Former systems based on Apache Hadoop MapReduce like Dedoop [Kol14] or a project of University of California, Irvine [VCL10] are batch systems. A static workflow has to be configured and executed on a large amount of input data. These batch workflows can take some minutes to hours depending on the chosen algorithms, the amount and shape of the input data and hardware. They are not designed to run on a single computer and make heavy use of the hard disk to store intermediate results, decreasing the performance. On the other hand, projects for single computers like Febrl are fast since they run in memory, but they cannot handle large amounts of data because of their limitations to a single machine. The approach of this thesis is to bring both sides together in one uniform framework, scalable from one to hundreds of nodes.

[4]GNU General Public License, Version 3.0

[5]`https://github.com/numbnut/sddf`

Design and implementation of the framework

The main task of this thesis is to design and implement the scalable duplicate detection framework. The two properties modularity and scalability are in the focus during the framework design.

Modular The framework is supposed to be modular to enable reusability. Modularity is achieved by adapting the duplicate detection process of Christen and implementing the “Pipes and Filter” [BMR+96] design pattern. A pipe is a reusable module with an in- and output type. It processes data of an input type and outputs the result. Pipes can be concatenated resulting in a pipeline which actually behaves like a pipe. A pipeline sequentially executes all contained pipes and produces a result of the specific output type. Pipes can be replaced by other pipes of the same in- and output types.

Scalable The overall process and the implementation should be done in a way to scale with the amount of input data and computation nodes. Each step in the process has to be analyzed with respect to its scalability.

How does the framework scale?

According to Spark’s distributed in memory collection called RDD, the intermediate results are kept in memory and are therefore ready to be reused directly without persisting and reloading them. This should speed up the whole duplicate detection process. Three main factors influence the runtime of the deduplication process. There are the parameterized algorithms, the cluster hardware setup and the shape and size of the input data. Every single factor can lead to a long runtime or even worse to an aborting process. The evaluation is done on a small cluster of up to 4 worker nodes and one master node. In the focus are time and space needed by the duplicate detection process, while varying the input set size and shape as well as the cluster size.

1.3. Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 covers the related work and outlines former publications about duplicate detection in general and especially parallel duplicate detection. Moreover, Apache Hadoop, Apache Spark and other related frameworks or projects are described. Chapter 3 describes the duplicate detection process and its adaption to a distributed environment. The next Chapter 4 details the architecture and usage of the implemented scalable duplicate detection framework SddF. Chapter 5 covers the evaluation and its experiments conducted to analyze and quantify the suitability of the implementation. Chapter 6 is about deliberation of the results and future prospects.

A List of all images, tables and listings can be found at page 74. A List of all acronyms can be found at page 71.

1.4. Demands on The Reader

The reader should have an understanding of computer science fundamentals like runtime complexity and other basic concepts. The reader should be familiar with the duplicate detection domain since not every detail is introduced in this thesis. Naumann and Herschel wrote a short book "An Introduction to Duplicate Detection" [NH10] which is a brief introduction to the duplicate detection domain. Christen's book "Data Matching" [Chr12] provides also an overview of this domain, but it is much more comprehensive and covers much more algorithms and subtopics. Beneficial would be to have basic experiences in object oriented and functional programming with Scala, since all examples are written in Scala.

1.5. Typographic Conventions

Any inline code snippet like class names and method names are formated with a typewriter font like `Classname`. A plural ending is also formated with typewriter like `Classnames` to improve the readability. Methods or functions are suffixed by brackets and left out parameters are replaced by dots like `println(...)`. The first occurrence of an acronym is written out.

2. Related Work

The duplicate detection problem is known by various names. Some of them are Record Linkage [BMC+03, FS69], Data Linkage [CCH04], (Near) Duplicate Detection [ACG02], Duplicate Elimination [ACG02], Deduplication [SB02], Merge-Purge [HS95], Identity Uncertainty [PMM+02], Citation / Reference Matching [PMM+02] and Approximate String Join [CGGM03]. To get an overview of existing projects and publications, a hierarchical classification of duplicate detection approaches is introduced, which is shown in Figure 2.1.

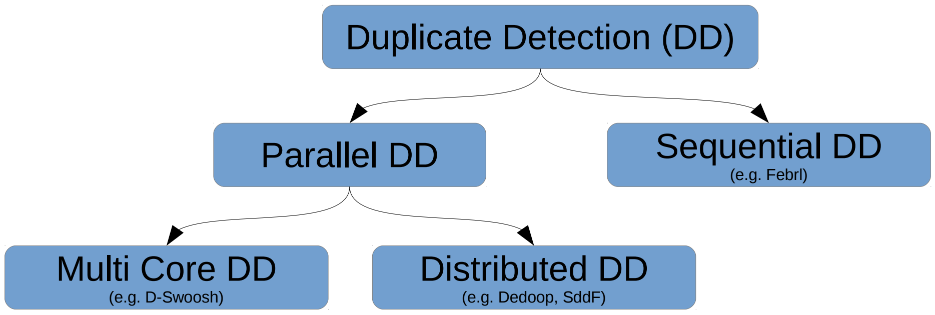


Figure 2.1.: Hierarchical classification of different duplicate detection approaches.

At first there is **Sequential Duplicate Detection** which does not take parallelization into account. **Parallel Duplicate Detection** covers all parallel approaches. Further Parallel Duplicate Detection can be divided into **Multi Core Duplicate Detection** and **Distributed Duplicate Detection**. Multi Core Duplicate Detection only deals with the parallelization on a single computer which is covered by multiple publications [KL07, BGG+06, BGH11, KL10]. The project D-Swoosh from the Stanford University is one of them. Distributed Duplicate Detection comprises the parallelization via a distributed system. Distributed Duplicate Detection is covered by multiple publications. The oldest project is GOMMA [KKH+10] from the University Leipzig, Germany. It is a distributed application using Java Remote Method Invocation (RMI). All other approaches use Hadoop MapReduce as distributed computing engine. In 2010, Vernica et al. [VCL10] implemented a duplicate detection process based on PPJoin+ [XWLY08] process using MapReduce. In 2012, the project Dedoop was started at the University of Leipzig, Germany. As its name suggests it combines deduplication and Apache Hadoop. Dedoop is detailed in Section 2.4.

This chapter starts with Section 2.1 about the duplicate detection process. The Sequential

Duplicate Detection project Febrl is introduced in Section 2.2 Also a short introduction to Apache Hadoop in Section 2.3 and to Apache Spark in Section 2.5 is given. An overview of other related projects can be found in the Appendix A.2 on page 78.

2.1. Duplicate Detection Process by Christen

The duplicate detection process [Chr12, p. 23] presented by Peter Christen is the base for the duplicate detection process used in this thesis. It is the theoretical foundation of this thesis and consists of six steps which are shown in Figure 2.2. First one is the data preprocessing. In this step, different data sources are aligned and formats are adjusted. Also activities to increase the data quality are applied in the preprocessing step. For instance the removal of invalid characters could be one preprocessing activity.

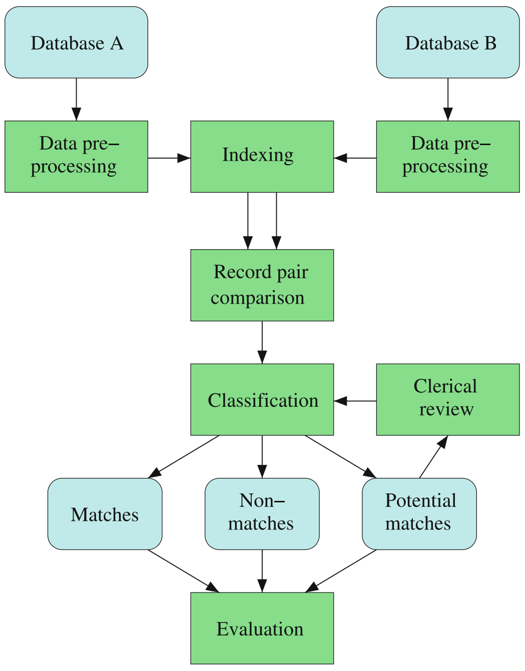


Figure 2.2.: Original duplicate detection process of Christen [Chr12, p. 24]

The next step is the indexing phase which is crucial because it determines the amount of tuple pairs to be analyzed. The amount of tuple pairs to be analyzed is called search space. The naive search space of n tuples is of the size of $\frac{n^2-n}{2}$ which is infeasible to process in the case of a large data set. Indexing algorithms are designed to reduce the size of the search space by multiple orders of magnitude in a subquadratic time complexity. Usually indexing algorithms are blocking algorithms which select subsets of the input data set, called blocks. In each block all tuples are compared pairwise. The reduced search space is the union of all tuple pairs contained in the blocks. Its size is much smaller than the naive

search space. After building the search space all contained tuple pairs [6] are compared featurewise by similarity measures. Similarity measures are often defined on strings, but there are also other possible input types like numeric types. Some similarity measures like the Edit Distance [Nav01] are metrics, but others, like the Jaccard Index [Chr12, p. 112], are not. Often they are wrongly named string metrics. Also phonetic encodings or other preprocessing steps can be included in the similarity calculation. Result of the similarity calculation of two k dimensional tuples is a similarity vector of size k.
The similarity vector is used in the classification phase to determine the membership of a tuple pair to one of the three classes match, non-match, potential match. All pairs classified as match are duplicates, non-matches are considered to be no duplicates. The class of potential matches are tuple pairs which can not be assigned to one of the two classes. These pairs are supposed to be classified manually in the clerical review step. There are two classes of classification algorithms. On the one hand, there are trained machine learning classification algorithms like Decision Trees [HK06, p. 291], Naive Bayes Classifier [VME03] or Support Vector Machines (SVMs) [HK06, p. 337] [Vap95, p. 138]. On the other hand, there are untrained algorithms like a threshold based classification. The classification result is the set of all tuple pairs labeled as a match which can be interpreted as a match graph. Every tuple represents a node and a match classification represents an edge between the tuples of the related pair.
The evaluation phase calculates evaluation measures like recall and precision to determine the performance of the process. To calculate these measures the gold standard for the input set is needed. The gold standard is the correct solution of the duplicate detection problem. It contains all true positive tuple pairs.
The result of the classification phase is not necessarily unambiguous [Chr12, p. 149]. Constellations like a is a duplicate of b and b is one of c, but a is not a duplicate of c are possible. To guaranty the absence of such transitive contradictions it is necessary to resolve them in a clustering step. A simple way to achieve this is to compute the transitive closure of all matches. It is possible to incorporate more advanced algorithms especially to avoid long duplicate chains or other cluster shapes with a large diameter. For more advanced clustering algorithms read the book "An Introduction to Duplicate Detection" [NH10]. Christen covers this topic, but does not include it in his duplicate detection process.

2.2. Febrl - Freely Extensible Biomedical Record Linkage

Febrl[7] is an open source duplicate detection software written in python. It is developed by Christen et al. at the Australian National University. The project started in 2004 and

[6] Christen uses the word record instead of tuple.
[7] `http://sourceforge.net/projects/febrl/`

the latest version 0.4.2 was released in 2011. Febrl implements the duplicate detection process of Christen described in 2.1. It is very feature rich and implements multiple blocking algorithms, similarity measures and classification algorithms. It offers a graphical user interface and the possibility to analyze the input data. The initial paper [CCH04] introduced Febrl as a multi threaded application using pypar[8]. This aspect seems to be dropped since there are no hints on running Febrl in parallel in the Febrl documentation. Also in his book “Data matching” [Chr12] he did not reference his software in the section “Parallel and Distributed Data Matching”. For this reason Febrl is categorized as a sequential duplicate detection software in this thesis.

2.3. Apache Hadoop

Apache Hadoop is an open source cluster computing framework and the de facto standard for big data analysis. An overview of Apache Hadoop and many of its extensions can be found in the literature [Hol15, SG14]. There are multiple projects build around Hadoop forming a wide ecosystem. Hadoop is written in Java. Since version 2.0 Apache Hadoop consists of three parts which are Hadoop Distributed File System (HDFS), Yet Another Resource Negotiator (YARN) and MapReduce. The division of the project into three parts opens the platform for other implementations of distributed computing frameworks like Apache Spark which may run on top of YARN replacing MapReduce. All three projects have a master slave architecture.
HDFS is a distributed, redundant, reliable storage layer. A HDFS cluster consists of a master called NameNode and a set of DataNodes. The NameNode is only responsible for managing meta data and request routing to the DataNodes. To minimize the probability of the master being a bottleneck all read and write queries are direct connections between client and DataNode. It is possible to configure a High Availability (HA) HDFS cluster with a NameNode failover. This architecture allows highly scalable cluster setups.
YARN is a resource manager for distributed computing. Since resource management is not in the scope of this thesis YARN, is not considered. Apache Hadoop MapReduce is a map reduce implementation as a YARN application.
Map reduce [DG04] is a parallel programming paradigm introduced by Google. It is inspired by the common functions `map(...)` and `reduce(...)` in functional programming. In the map phase all input entities are mapped to a key value pair. Afterwards the reduce step reduces all pairs sharing the same key to a single value. A word count example is depicted in Figure 2.3. In this case, the keys are the words in the source document and the values are initialized with a constant 1. The reduce function is an addition, which sums up the occurrences of every single word. The architecture of an Apache MapReduce cluster is similar to the HDFS architecture. A master called JobTracker handles the job

[8]Pypar is a message driven parallelization framework which uses the Message Passing Interface (MPI).

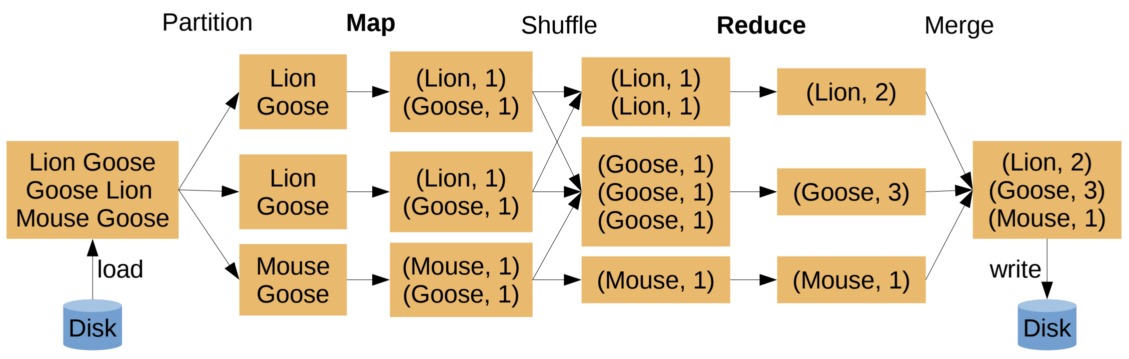


Figure 2.3.: Word count example using the map reduce paradigm.

execution requests of a client. It locates the involved data and a TaskTracker with free slots near the data. The JobTracker extracts the map and reduce tasks of the job and sends them to the TaskTrackers. The TaskTrackers are the workers which execute these tasks. The results are stored in the HDFS.

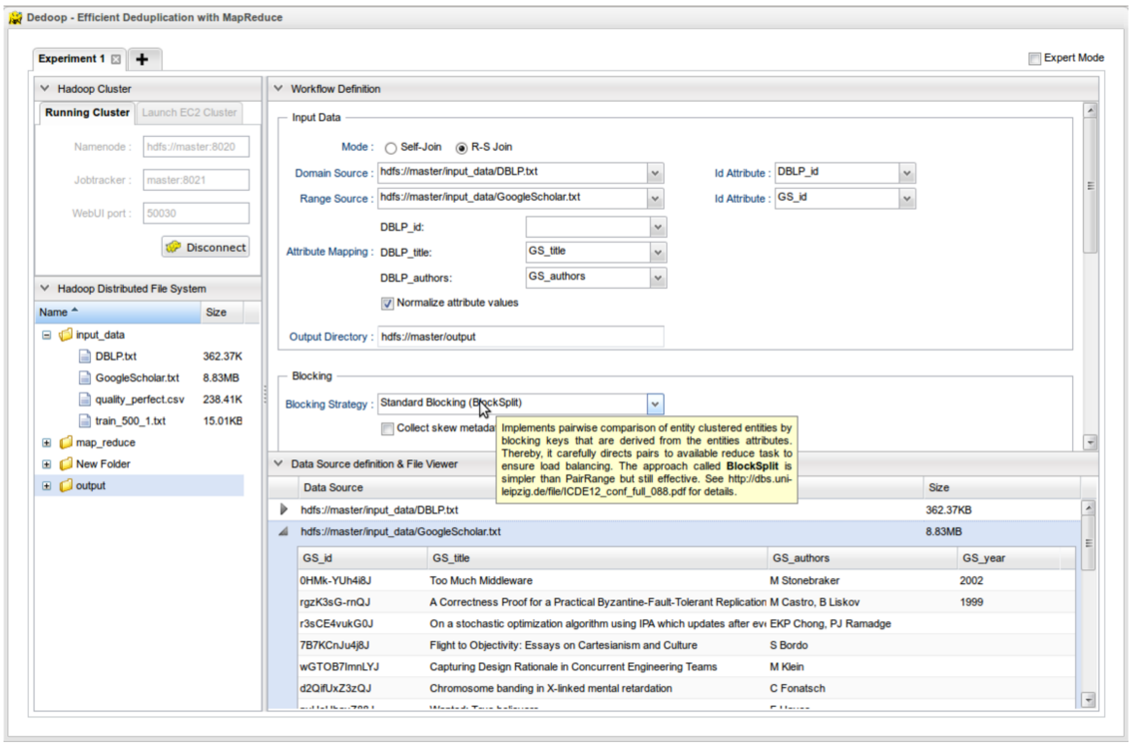
There are some drawbacks in the Hadoop MapReduce architecture. All results of a map reduce step are persisted and afterwards read from disk again by the next map reduce step. Performance degrades especially in the case of iterative algorithms. Another restriction is the map reduce pattern itself. All algorithms have to be adapted to this pattern which can be difficult and time consuming. Using the MapReduce Application Interface (API) results in many lines of code. Even a word count example is about 47 lines long. The listing can be found in the Appendix A.1 on page 79. Many projects aim to relax these restrictions. There are high level languages like Pig Latin which is a procedural language on top of MapReduce. The Hive Query Language (HQL), which is included in Apache Hive, a data warehouse framework, is a declarative SQL-like language. The project iMapReduce [ZGGW11] extends Hadoop MapReduce and tries to optimize it for iterative algorithms. With iMapReduce it is possible to write an iterative MapReduce job, which don't have to persist intermediate results. Apache Mahout[9] is a machine learning library for Apache Hadoop. At the moment, the Mahout project migrates to the Apache Spark platform.

2.4. Dedoop - Deduplication utilizing Hadoop MapReduce

Dedoop is a project of the University of Leipzig. Its goal is to solve big data duplicate detection problems. Like the name suggests it uses the Hadoop platform as backend. The project is part of the dissertation [Kol14] of Lars Kolb at the University of Leipzig. This work focuses on adapting blocking algorithms to the map reduce paradigm. Another topic is load balancing of the different algorithms. Result of the work are several algorithms

[9]https://mahout.apache.org/

which are optimized for the map reduce paradigm. Dedoop offers a web frontend to configure and execute the process, which is depicted in Figure 2.4.



Namenode : hdfs://master:8020
Jobtracker : master:8021
WebUI port : 50030

Name	Size
input_data	
DBLP.txt	362.37K
GoogleScholar.txt	8.83MB
quality_perfect.csv	238.41K
train_500_1.txt	15.01KB
map_reduce	
New Folder	
output	

Mode : Self-Join [✓ R-S Join]
Domain Source : hdfs://master/input_data/DBLP.txt   Id Attribute : DBLP_id
Range Source : hdfs://master/input_data/GoogleScholar.txt   Id Attribute : GS_id
Attribute Mapping : DBLP_id:
DBLP_title: GS_title
DBLP_authors: GS_authors
✓ Normalize attribute values
Output Directory : hdfs://master/output

Blocking Strategy : Standard Blocking (BlockSplit)
Collect skew metada

Implements pairwise comparison of entity clustered entities by blocking keys that are derived from the entities attributes. Thereby, it carefully directs pairs to available reduce task to ensure load balancing. The approach called **BlockSplit** is simpler than PairRange but still effective. See http://dbs.uni-leipzig.de/file/ICDE12_conf_full_088.pdf for details.

Data Source	Size
hdfs://master/input_data/DBLP.txt	362.37KB
hdfs://master/input_data/GoogleScholar.txt	8.83MB

GS_id	GS_title	GS_authors	GS_year
0HMk-YUh4i8J	Too Much Middleware	M Stonebraker	2002
rgzK3sG-rnQJ	A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication	M Castro, B Liskov	1999
r3sCE4vukG0J	On a stochastic optimization algorithm using IPA which updates after ev	EKP Chong, PJ Ramadge	
7B7KCnJu4j8J	Flight to Objectivity: Essays on Cartesianism and Culture	S Bordo	
wGTOB7ImnLYJ	Capturing Design Rationale in Concurrent Engineering Teams	M Klein	
d2QifUxZ3zQJ	Chromosome banding in X-linked mental retardation	C Fonatsch	

Figure 2.4.: Screenshot of Dedoop User Interface

Dedoop is restricted to batch processing and all algorithms have to be adapted to the map reduce paradigm. It is built for clusters only and not usable on a single machine. Unfortunately Dedoop is closed source and there is no documented API released.

2.5. Apache Spark

Apache Spark is a cluster data processing framework and the official successor of Apache Hadoop. The first publication about Spark was made in 2009 and the first API stable version 1.0 was released in 2014. Apache Spark is a top level project of the Apache Software Foundation since February 2014. It is licensed under the Apache License 2.0. The project is maintained by UC Berkeley, a spin-off called Databricks and the community. The framework is introduced on the project website[10] as follows. “Apache Spark is a fast and general engine for large-scale data processing.” Its key features are distributed in-memory data structures, compatibility to the Apache Hadoop ecosystem, a lean programming model and its interactive shell. Accessible data sources are HDFS, HBase, Cassandra and any other Hadoop data source. Apache Spark can be deployed in standalone mode,

[10]`https://spark.apache.org/`

on YARN, Mesos or Amazon Elastic Compute Cloud (EC2). The project is written in Scala[11] and there are bindings for Java and Python. Another feature is the Spark Shell, which allows interactively executing code on the spark cluster. Besides the core API, Spark offers the four libraries Spark SQL, Spark Streaming, MLlib (machine learning) and GraphX (graph processing). The information presented in this section is from the official Spark documentation[12] and the book "Learning Spark" [KKWZ15].

2.5.1. Resilient Distributed Dataset (RDD)

The main concept of the Apache Spark core is the Resilient Distributed Dataset (RDD) [ZCD+12]. An RDD is a fault tolerant, distributed, generic collection. RDDs are immutable. Every method invocation on an RDD creates a new RDD to avoid side effects.
Each RDD consists of m partitions each residing on a cluster node. An RDD may have more or less partitions than the number of nodes in the cluster. The default partitioner is a `HashPartitioner`. On a cluster with n nodes the hash partitioner generates an integer hash h from 1 to m for each element. The element with hash h is assigned to partition h. Presuming a balanced hash function the elements are equally distributed through out the cluster. A `RangePartitioner` is also available which presumes an ordering on the input data set. The `RangePartitioner` splits the input data set into fixed sized ranges according to the ordering. This partition strategy is more expensive because it needs to sort the data. Maybe there is a more efficient approach to achieve a partitioning according to the ordering than sorting, but in every case it is more expensive than randomly assigning the elements to a partition by hashing.
The minimal number of partitions depends on the configuration. The default number of partitions is 2, which is not suitable in most cases. The spark tuning guide[13] recommends $2-3$ tasks per core. One task will be started for each partition. For that reason the number of partitions is an important parameter to operate at full capacity.
Fault tolerance is achieved by the concept of lineage. The lineage of an RDD is the dependence of the RDD on other parent RDDs which are necessary to derive the new RDD from. The operations performed on an RDD to create a new RDD produces a Directed Acyclic Graph (DAG). The lineage of a single RDD is the subgraph including all paths from data sources to the RDD. If a node crashes the lineage is used to recompute the relevant RDDs or partitions.
Spark divides the heap into two areas. The first one is the default storage of all RDDs. The second one is only used if an RDD is explicitly cached. The default size of the cache is

[11]Scala is a programming language for the Java Virtual Machine (JVM) which combines functional and object oriented paradigms. Since it compiles to Java bytecode, classes written in Scala can be accessed from Java and vice versa.
[12]`https://spark.apache.org/docs/1.3.1/`
[13]`https://spark.apache.org/docs/1.3.1/tuning.html`

60% of the heap space. The size of this cache is fixed, but it can be configured. This may lead to a waste of memory. The lineage is also used to handle cache misses. If an accessed RDD partition is not in memory anymore because it was replaced by a newer one, it is recomputed by traversing the DAG back to all needed partitions in order to reapply all operations on them. Figure 2.5 shows an example DAG. RDD E is the result of a join operation of C and D. C is a subset of A created by filtering and D results from grouping B by key. `Filter` is a map operation which maps each partition to a new one without shuffling data. `GroupByKey` and `join` needs to shuffle the data to build new partitions. The lineage of E is the whole DAG shown in Figure 2.5. For C the lineage is the subgraph consisting of A and C. For D the lineage is the subgraph consisting of B and D.

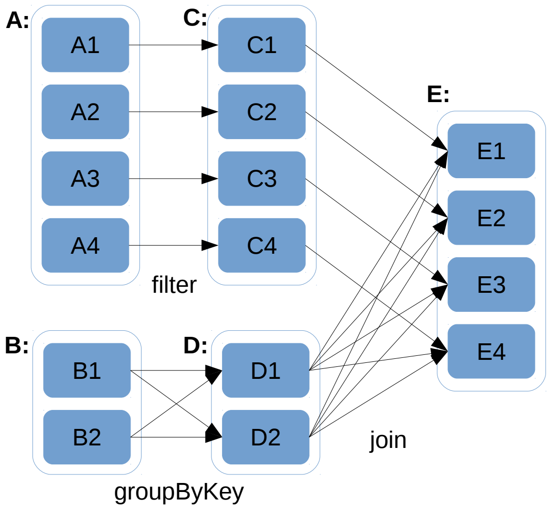


Figure 2.5.: Example Lineage of an RDD join. A, B, C, D and E are RDDs. A consists of 4 partitions A1 to A4.

One major advantage of Apache Spark compared to Hadoop is to have complete control over the persistence of data sets. It is possible to choose between different storage levels. The storage medium disk or memory can be chosen. Also a swapping-like level can be chosen, which will spill data to disk, if the application runs out of memory. It is possible to store objects serialized, which is in most cases more space efficient. The level of redundancy of each partition can be adjusted to minimize the impact of node crashes. The default storage level is deserialized in memory and the default redundancy is none.

MLlib

MLlib is the machine learning library built atop the Spark core. Amongst others, it provides clustering, classification and recommendation algorithms. For this thesis the three classification algorithms Naive Bayes, Decision Tree and Support Vector Machine can be used to classify candidate pairs. The provided clustering algorithms like K-Means are not suitable for the kind of clustering needed in the duplicate detection domain.

GraphX

GraphX is the graph computing library built atop the Spark core. It tries to be data centric and graph centric. That means a graph can be accessed as a collection of edge triplets or can be traversed like a graph. The graph model of GraphX is a property graph which is a generic directed multigraph. Properties are user defined objects which can be attached to vertices and edges. GraphX is used to build the match graph and resolve transitive contradictions.

2.5.2. Cluster Setup

A Spark cluster always consists of a cluster manager and n executors, which are all processes. In most cases a distributed file system like HDFS or a distributed database like HBase serves as a storage layer. If that is the case the storage cluster is typically aligned to the Spark cluster to exploit the locality of the data. It is also possible to access the file system of the server directly, but in that case the file paths have to be valid on all nodes. The Spark cluster can also be managed by YARN or Mesos. This is necessary if other cluster frameworks run in parallel on the same cluster. A basic cluster setup in standalone mode consists of a master which is the cluster manager and n worker nodes each running an executor process. Figure 2.6 shows this setup. In the first step the driver requests free executors from the master. Afterwards the driver process executes the user program, does the scheduling and submits the tasks to the executors. The scheduler takes the locality of the data into account. Spark will always try to process the data on the nodes where it resides. The guideline is “ship code not data”.

2.5.3. Code Example and Execution

Writing a driver program is not difficult. Listing 2.1 shows the “hello world” example of data processing, a word count. Everything needed to build and execute it is the Apache Spark core library and a Scala compiler. To interact with a Spark cluster the driver program has to instantiate a `SparkContext`. The `SparkContext` establishes the connection to the cluster and is used to create RDDs. Afterwards the example input data set is read from the file “/lorem-ipsum.txt”. The method `textFile` creates a `RDD[String]` which is a collection of all lines in the file. The method `flatMap` also creates an RDD[String] which is a collection of all words in the file. To count the words each word is expanded to a pair of the word itself as key and a counter initialized with one as value. The method

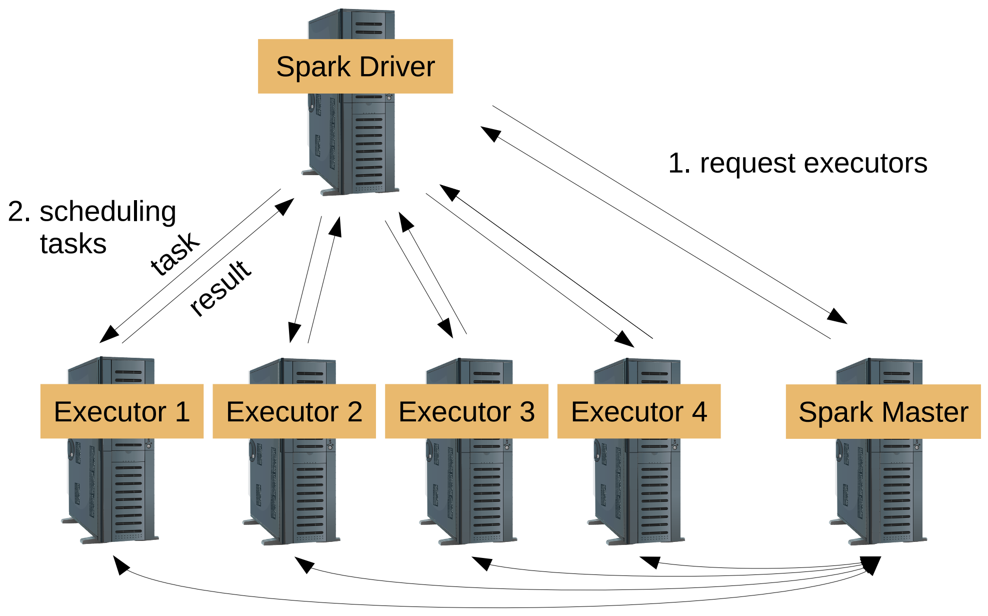


Figure 2.6.: Basic Spark cluster setup in standalone mode with HDFS.

`reduceByKey` applies the passed function to all values with the same key. This will sum up all counters of the same word which will result in a key value pair of the word and the number of its occurrences in the file.

```scala
package de.unihamburg.vsis.sddf.examples
// imports left out
object WordCount extends App {
  val conf = new SparkConf()
    .setAppName("WordCount").setMaster("local")
  val sc = new SparkContext(conf)
  // hdfs://master:9000/test.txt
  val file = sc.textFile("/lorem-ipsum.txt")
  val words = file.flatMap(line => line.split(" "))
  val wordNumbers = words.map(word => (word, 1))
  val counts = wordNumbers.reduceByKey(_ + _)
  counts.foreach(println(_))
}
```

Listing 2.1: Word count example using Apache Spark. Imports are left omitted.

There are different ways to execute a Spark application. The easiest way which is very helpful during development is the local mode. If the Spark master is set to local, a minimal Spark cluster with only one executor is started on the local machine. In this case, the driver and executor run in the same JVM, speeding up the Spark application. The driver connects to the local executor and submits jobs and receives results. The local mode enables a fast development workflow because there is no need to deploy and start the

application on a real cluster. Instead it can be executed on the developer's machine or even from the Integrated Development Environment (IDE).
For a submission of an application the script `spark-submit` is the tool. There are several parameters which influence the execution of the application. The application jar needs to be passed to spark-submit. The parameter "--master" sets the master URL and "--class" sets the class containing the main method which is executed. The memory per executor can be adjusted with the parameter "--executor-memory". Arbitrary configuration properties can be set via "--conf <key>=<value>". The "--deploy-mode" can be set to client or cluster. If client is chosen the driver application will be executed on the local machine. In cluster mode the application is shipped to the cluster and executed directly on the cluster. The cluster mode avoids network latency which can be a relevant factor. The following lines show an example invocation of the spark-submit script to start the word count example compiled and packaged into the hello-world.jar.

```
spark-submit --executor-memory 6G\
    --class de.unihamburg.vsis.sddf.examples.WordCount\
    --master spark://master:7077 hello-world.jar
```

Listing 2.2: Example submission of an application to a spark cluster.

3. Deduplication Process

This chapter covers the adaption of the process of Christen introduced in Section 2.1. Four changes have been made to the process model of Christen. To simplify the process only the self join of a single input data set is considered. Christen uses two input sets. In our process the reading of the gold standard is a separate step. Our process does not include explicitly potential matches, which require a clerical review. Such a feedback is covered by the classification step and would be handled internally without affecting other steps. Christen does not depict the clustering step in his process, but mentions it in a subsequent chapter. We added it explicitly to our process. An overview of our process steps and its intermediate result types is shown in Figure 3.1. This chapter focuses on how much overhead does the parallel execution of each step on the cluster produce or in other words how does each step scale horizontally?

3.1. Reading and Preprocessing

Reading or parsing is the process of extracting tuples from the input sources. A common case is to map each line of a file to a single tuple which is identifiable by an unique id. A tuple is a fixed set of features. The feature names are defined by the tuple schema which is identical for all tuples in the same duplicate detection process. Because all tuples share the same schema, features can be empty, which means they contain a null value. An example schema and tuple instances could look like the following ones.

```
Schema: (id, forename, lastname, birthdate, place-of-birth)
----------------------------------------------------------
Tuple1: (1, Max, Mustermann, 01.01.1970, )
Tuple2: (2, Silva, Hansen, 31.12.1968, Brüssel)
```

Assuming a sharded file in a distributed filesystem the parsing can be done in parallel. Scalability of the parsing step actually depends on the locality of the input data. Further we will assume a sharded input file which is equally distributed among the cluster nodes. In this environment the parallel parsing does not produce any overhead.
Preprocessing is the step to increase the data quality. Assuming an isolated preprocessing

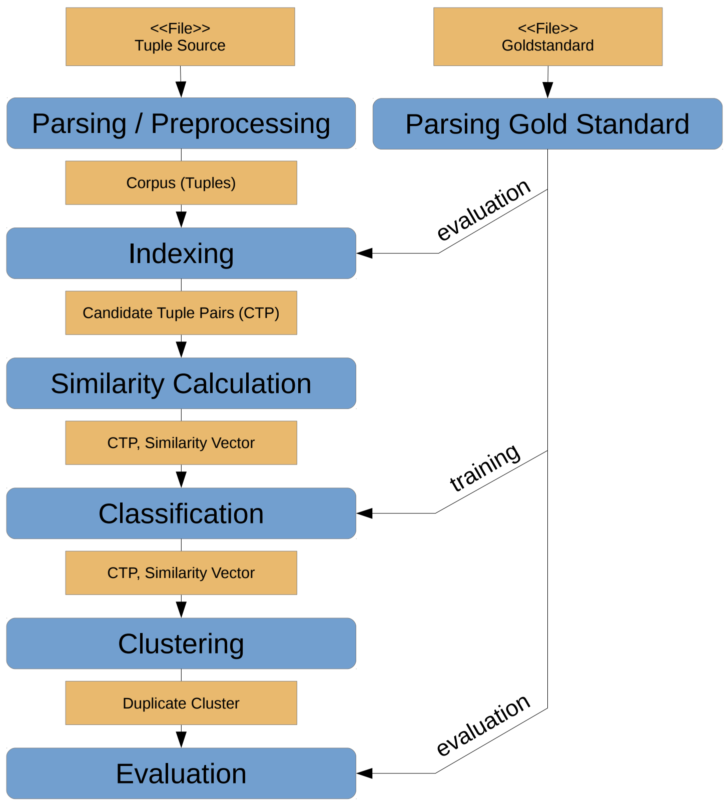


Figure 3.1.: Overview of the deduplication process and its intermediate result types.

which means the preprocessing of a tuple doesn't depend on other tuples, preprocessing can be done in parallel without any overhead. Common preprocessing steps like trimming and regular expression removal are isolated because they do not have any dependencies to other tuples. Under the mentioned preconditions the parsing and preprocessing is not a crucial step and scales without any overhead. The set of tuples, resulting from the preprocessing step, is called corpus.

3.2. Indexing

Indexing is more complex than the first step. Indexing algorithms are used to reduce the quadratic search space. We focus on blocking algorithms which produces sets of tuples called blocks. All possible tuple combinations of two tuples in a block are candidate tuple pairs. The candidate tuple pairs of all blocks combined are the reduced search space. All pairs not contained in a block are supposed to be non-matches. If there are true duplicate tuple pairs not contained in a block, the pair is lost for further processing. For that reason blocking is a trade off between the size of the search space and the maximum recall. The runtime behavior depends on the algorithm. We want to discuss three blocking algorithms which will be shortly introduced at first. All of them have the first step in common. Every tuple of the corpus C is mapped to a blocking key which is a combination of parts of features. The performance of the algorithms depends on the choice of the blocking key.

3.2.1. Standard Blocking

Standard Blocking [FS69, p. 64] constructs an inverted index which maps a blocking key to a set of corresponding tuples all sharing this blocking key. Each set is a result block. The following listing shows an implementation of Standard Blocking using Scala and the Spark API.

```scala
val bkvTuplePairs: RDD[(String, Tuple)] = input.map(
t => (bkvBuilder.buildBlockingKey(t), t)
)
val keyBlocks: RDD[(String, Iterable[Tuple])] = bkvTuplePairs.groupByKey
val blocks: RDD[Seq[Tuple]] = keyBlocks.map(_._2.toSeq)
```

The crucial operation is the `groupByKey(...)` which shuffles the data according to the blocking key. The best case is to have all tuples sharing a blocking key colocated at the same node which does not need any shuffling. In the worst case, all tuples share the same blocking key and the tuples are distributed equally among k nodes. This scenario needs to shuffle $|C| - \frac{|C|}{k}$ which is nearly $|C|$ for large clusters. We can infer the upper bound of $|C|$ of data to be shuffled. The actual amount of data to be shuffled depends on the choice of the blocking key and the distribution of the data. A possible optimization would be to use a partitioner which partitions the data by the blocking key. In this case it is not necessary to shuffle data. This only works if the data is partitioned by the blocking key or if the data is fully replicated.

3.2.2. Sorted Neighborhood

Sorted Neighborhood [HS95, HS98] sorts the tuples by their blocking keys. Afterwards a sliding window of a fixed size s is applied to the tuple list. Every possible position of the window results in a block of s tuples. Neighboring blocks are overlapping by $s - 1$. The crucial operation is the sorting of the data. According to “Minimal MapReduce Algorithms” [TLX13] time complexity of distributed sorting using TeraSort comes near the time complexity $O(n \cdot log(n))$ on a single machine. Having the data partitioned equally among the k cluster nodes the fraction of $\frac{k-1}{k}$ of the corpus has to be shuffled. For large cluster sizes $\frac{k-1}{k} \approx k$ which implies the upper bound of $|C|$ tuples. To achieve a correct sliding window on a partitioned collection the partitions need to overlap by $s - 1$ tuples. This increases the partition size by $s - 1$. If $s << \frac{|C|}{k}$ the increase of the partition size is very small and can be ignored. Thus, like Standard Blocking, Sorted Neighborhood is also well suited for distributed environments.

3.2.3. Suffix Array Blocking

Suffix Array Blocking [AO05] is a special case of Standard Blocking as presented in Section 3.2.1. The blocking key of a tuple is extended to a set of blocking keys which is the set of all suffixes of the blocking key. Every tuple is inserted into each block corresponding to one of the blocking keys in its blocking key set. Two parameters influence the result of the algorithm. On the one hand, there is the minimum suffix length $minsl$ which restricts the blocking keys derived from the original blocking key to a minimum length. On the other hand, there is the maximum block size $maxbs$ which drops every block whose size exceeds the maximum block size.
In general the conclusions made at Standard Blocking can be transfered to Suffix Array Blocking considering the blocking key set. To estimate an upper bound of the shuffled data we need to know the maximum blocking key length b_{ml}. In the worst case every blocking key in each set needs to be shuffled which implies an upper bound of $|C| \cdot b_{ml}$. This upper bound is a coarse estimation as it implies $minsl = 1$ and $maxbs = \infty$ which is the worst case and an unrealistic configuration. In a realistic application of the algorithm the amount of data to be shuffled is much smaller.

3.3. Similarity Calculation

The two tuples of each tuple pair are compared featurewise by a similarity measure which results in a similarity vector. A similarity measure $sim(a, b)$ is a function comparing two arbitrary strings a and b and returning a similarity value $s \in [0, 1]$. The similarity value 1 stands for absolutely similar and 0 for absolutely dissimilar. There are different similarity measures available like the Jaccard Index [NH10, p. 24]. A second class of functions, which can be applied to measure similarity, are distance functions. A distance function $dist(a, b)$ assigns a distance $d \geq 0$ to a pair of strings a and b. Typically, distance functions return an integer value like the Levenshtein Distance [NH10, p. 30]. Distance functions can be normalized to the range $[0, 1]$ using the length of the compared strings, if the distance is limited by the maximum string length $max(|a|, |b|)$. The normalization combined with an inversion of the range turns a distance function into a similarity measure.

$$sim(a, b) = 1 - \frac{dist(a, b)}{max(|a|, |b|)} \tag{3.1}$$

For most measures the similarity calculation can be done completely in parallel, since the calculation of the similarity vector only depends on the tuple pair.

3.4. Classification

The classification step can be done completely in parallel. When using an untrained classification model like a threshold-based approach every tuple can be processed in parallel. This also holds for a trained model, but the training of this model must be considered. There are three different ways to train a machine learning model. Assuming the amount of training data fits into the memory of a single node the model can be trained on a single node and afterwards it gets distributed to the other nodes. This approach can be applied to all machine learning algorithms. If the training data is too large for a single machine or if the process performance needs to be improved it is possible to train the model in a distributed manner. Two different methods are distinguished. The first one can be applied to all machine learning algorithms. Each node trains its own model. Afterwards a fixed odd number n of models is selected and distributed throughout all nodes. Each tuple gets classified by all of the selected models. A majority voting is done which returns the final classification result. The runtime of this approach is n times longer since n classifications have to be done. The second approach only addresses mergeable models. Each node trains it own model and than they are merged into one final model which is distributed throughout all nodes. Hall et al. published an approach to combine decision trees learned in parallel [HCB98].

3.5. Clustering

The classification result, consisting of matching tuple pairs, can be interpreted as an undirected matching graph $M = (V, E)$. The vertices V are all tuples in a match relation and the edges E are all tuple pairs classified as a match. The sizes of V and E depend on the size and shape of the data and the configuration of the classifier. For instance, a low threshold may increase the amount of tuples in the match relation and additionally increases the amount of edges, which correlates with a bad precision.
The result of the classification step may contain contradictions, like Christen detailed out in [Chr12, p. 149]. Having two matches (t_a, t_b) and (t_a, t_c) and a non-match (t_a, t_c) is such a case. Matching subgraphs with a large diameter like long chains $(t_0, t_1) \dots (t_{i-1}, t_i)$ are a problem, since (t_0, t_i) may be classified as a non-match very likely.
The clustering is the step to resolve contradictive duplicate relationships in the classification result. One way to achieve this is to compute the connected components of the graph and assigning all tuples in a component to one real world entity. This approach is often also referred to calculating the transitive closure. There are other approaches like Center Clustering [HM09], which tries to build compact clusters with a small diameter and a high similarity. Center Clustering needs to process every subgraph sequentially, which makes it a sequential algorithm in general, since the worst case is only one single

subgraph. However, if the classification highly partitions the resulting match graph into multiple connected components, a parallel execution of Center Clustering is possible.
In general, computing the connected components is a complex operation. Using depth-first search the worst case runtime to compute the connected components is $O(|V| + |E|)$ [Gol80, p. 41]. The runtime complexity seems to be linear, but that is not the case. In a fully meshed graph the runtime is dominated by $|E| = |V|^2$, which results in a quadratic runtime complexity $O(|V|^2)$. The advantages of the match graph are, that it should be sparsely connected and the connected components should be small. Having a fully meshed graph does not fit into the domain of duplicate detection. Also large connected components dominating the match graph are supposed to be artifacts which should be avoided by optimizing the classifier. These two assumptions speed up the calculation in practice. Nevertheless, the clustering step is the step with the highest theoretical time complexity.

3.6. Evaluation

The two common evaluation measures of a duplicate detection result are recall and precision [NH10, p. 61] [Chr12, p. 167]. The main question of this section is how these measures can be computed on a partitioned distributed result R without shuffling tuples across the network. Figure 3.2 shows how a duplicate detection result R is separated into its four components. The result itself consists of the true positives TP which are duplicate matches according to the gold standard G and the false positives FP which are not element of the gold standard. The false negatives FN are all tuple pairs contained in the gold standard G, but not in the result R. All other tuple pairs are true negatives TN because they are correctly identified to be non-matches.

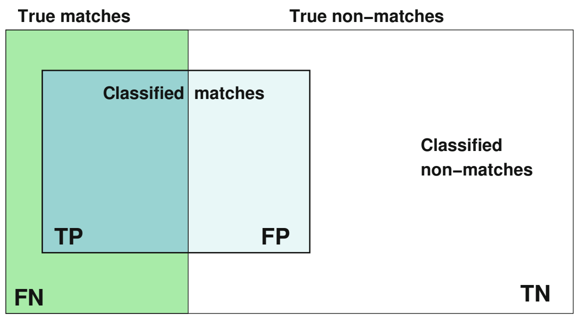


Figure 3.2.: This graphic shows the relevant sets to evaluate a duplicate detection result. Goal is to align the set of classified matches with the one of the true matches. That would minimize the amount of false positives FP and false negatives FN to zero and maximizes the true positives TP and true negatives TN. [14]

The recall depends on the result R and the gold standard G, see Equation 3.2. It states which fraction of the desired result set (gold standard) is included in the result. The

[14]Source: P. Christen, Data Matching [Chr12, p. 166]

precision also depends on the result R and the gold standard G, see Equation 3.3. It states the fraction of correctly identified tuple pairs in the result.

Recall:

$$rec = \frac{|G \cap R|}{|G|} = \frac{|TP|}{|TP \cup FN|} \tag{3.2}$$

Precision:

$$prec = \frac{|G \cap R|}{|R|} = \frac{|TP|}{|TP \cup FP|} \tag{3.3}$$

Given the gold standard G and the result R the following equations 3.4 to 3.6 are valid. The true negatives TN are not relevant for our calculations.

$$TP = G \cap R \tag{3.4}$$

$$FN = G - R \tag{3.5}$$

$$FP = R - G \tag{3.6}$$

In our case, the result R is partitioned into multiple disjoint result sets which reside on the cluster nodes. A naive calculation of these measures is expensive because tuple pairs needs to be shuffled across the network. How can the calculation of these measures be extended to a distributed result $R = \bigcup_{i=1}^{n} R_i$ with $R_a \cap R_b = \emptyset \wedge a \neq b$?
Both recall and precision can be calculated much more efficiently than using high level set operations on the partitioned result. At first it is analyzed how the recall can be computed partially on every node and how these partial results can be aggregated to an overall recall. The following assumption can be made. The gold standard G is available on all nodes or at least fast accessible on every node. The result partition R_n which is disjoint to all other partial results resides on the cluster node n. Equation 3.7 shows how the recall is adapted to a partitioned result set.

$$\begin{aligned} rec_{distr} = \frac{|TP|}{|G|} &= \frac{|G \cap R|}{|G|} = \frac{|G \cap (\bigcup_{i=1}^{n} R_i)|}{|G|} = \\ \frac{|\bigcup_{i=1}^{n} (G \cap R_i)|}{|G|} &\underset{R_a \cap R_b = \emptyset,\, a \neq b}{=} \frac{\sum_{i=1}^{n} |G \cap R_i|}{|G|} = \sum_{i=1}^{n} \frac{|G \cap R_i|}{|G|} \end{aligned} \tag{3.7}$$

$\frac{|G \cap R_i|}{|G|}$ is computed on every node locally. Afterwards the partial results are added up to calculate the overall recall.
Equation 3.8 shows how the precision is adapted to a partitioned result set.

$$prec_{distr} = \frac{|TP|}{|TP \cup FP|} = \frac{|G \cap R|}{|R|} = \frac{|G \cap (\bigcup_{i=1}^{n} R_i)|}{|\bigcup_{j=1}^{n} R_j|} = \frac{|\bigcup_{i=1}^{n}(G \cap R_i)|}{|\bigcup_{j=1}^{n} R_j|} \underset{R_a \cap R_b = \emptyset,\, a \neq b}{=} \frac{\sum_{i=1}^{n} |G \cap R_i|}{|\bigcup_{j=1}^{n} R_j|} = \frac{\sum_{i=1}^{n} |G \cap R_i|}{\sum_{j=1}^{n} |R_j|} \tag{3.8}$$

To calculate the precision of a partitioned result set $R = \bigcup_{i=1}^{n} R_i$, $|R_i|$ and $|G \cap R_i|$ have to be calculated for every partition. Afterwards the results are aggregated to the overall result.
Distributing the gold standard throughout all cluster nodes can be a limitation. Especially if the gold standard consumes a large fraction of a cluster nodes memory or if it does not fit into the nodes memory. If both, the gold standard G and the resulting R partitions, are aligned the calculation of recall and precision can be done directly on the partitioned sets without any communication overhead. Aligned means only partitions R_i and G_i on the same node i have a non empty intersection. All other combinations have an empty intersection $G_i \cap R_j = \emptyset \wedge i \neq j$. If R and G are aligned, the following equation holds.

$$G \cap R_i \underset{G_i \cap R_j = \emptyset, i \neq j}{=} G_i \cap R_i \tag{3.9}$$

The alignment is done by using the same hash partitioner for R and G. If both sets are partitioned on basis of the tuple id pairs, the same pairs will reside on the same node. This leads to the new equations for precision Equation 3.10 and recall Equation 3.11, which are computed with a partitioned, but aligned gold standard.

$$prec_{distr} = \sum_{i=1}^{n} \frac{|G_i \cap R_i|}{|G|} \tag{3.10}$$

$$rec_{distr} = \frac{\sum_{i=1}^{n} |G_i \cap R_i|}{\sum_{j=1}^{n} |R_j|} \tag{3.11}$$

The F-measure is also a widely used evaluation measure [NH10, p. 62] [Chr12, p. 168]. It is the harmonic mean of precision and recall. The F-measure shown in Equation 3.12 can also be calculated in a distributed manner, since it only depends on the recall and the precision. The same preconditions and restrictions hold for the F-measure.

$$fmeas = 2 \cdot \frac{prec \cdot rec}{prec + rec} \tag{3.12}$$

4. SddF - Scalable Duplicate Detection Framework

The two main requirements described in Section 1.2 are modularity and scalability. The requirement of modularity is driven by two motivations. It should be easy to contribute new algorithms and it should be possible to recombine the modules in a different way to enable the user building new data processing workflows. All modules and interfaces are supposed to be designed scalable. Scalable in this case explicitly means scaling up to hundreds of nodes and scaling down to a single node.

4.1. Pipelining

To achieve a modular and lean design we make use of the “Pipes and Filter” design pattern [BMR+96, p. 53]. Our wording differs from the original one chosen by Buschmann et al.. Pipes in our case are only defined by an input and an output type. They do not exist in form of a class. In this thesis filters are called pipes because we think the semantics of a filter does not cover all possible pipes. A pipe could also be a transformer and in that case the name filter could be irritating.
Many other frameworks mentioned in Section 4.1.1 make use of this pattern or similar concepts. Unfortunately, all listed frameworks do not meet the requirements except for the Spark Dataflow project. The project was started during the creation time of this thesis thus it has not been considered. Therefore, it seems to be the best solution to implement an own lightweight pipelining framework. The foundation of all process steps of the duplicate detection process is a small pipelining framework called PipE which is described in Section 4.1.2.

4.1.1. Related Work

In this section different pipelining frameworks or related projects are introduced and discussed.

Apache Spark ML Pipelining API

The Spark ML Pipelining API was introduced with the 1.2 release of the MLlib and is the new standard to which all algorithms are migrated. The ML `Pipeline` consists of a List of `PipelineStages` which can either be `Estimators` or `Transformers`. An `Estimator` is a machine learning algorithm which derives a `Model` from the input data. A `Transformer` is a simple stage of a `Pipeline` which transforms an input `SchemaRDD` to an output `SchemaRDD`. A `Model` is a `Transformer`. The MLlib `Pipeline` is restricted to operate on `SchemaRDDs`. A `SchemaRDD` acts like a table in which columns and rows can be added. A `Transformer` can modify the table by adding, updating, or deleting rows or columns. The user has to ensure the right order of the `PipelineStages`. There is no kind of typechecking whether a specific column is present during compilation time. A noteable feature of the ML Pipelining API is the possibility to do automated parameter optimization. During the implementation of our framework the ML Pipelining API has not been released yet. Also the restriction to `SchemaRDDs` lead to the decision to avoid using the ML Pipelining API. Further information can be obtained in the chapter "Pipeline API" in the book "Learning Spark" [KKWZ15, p. 236].

Spark Dataflow

Spark Dataflow[15] is a prototypical implementation of the Google Cloud Dataflow (GCD) execution backend connector. Every Pipeline written against the GCD pipelining API can be executed on a Spark cluster.
GCD emerged from the FlumeJava [CRP+10] project which covers pipelining and pipeline optimization atop on MapReduce. It is designed for parallel batch processing. The core concept of FlumeJava is the `PCollection` which is a parallel collection. A `PCollection` provides methods for distributed processing like `parallelDo()` which is like a `map()` and a `combineValues()` corresponding to `reduce()`. For key value pairs FlumeJava provides a special class called `PTable`. `PTable` provides special functionality for key value pairs like `groupByKey()`. FlumeJava does a deferred evaluation of the pipeline which improves optimization possibilities. The FlumeJava API looks very similar to the one Apache Spark offers.
Another Google project is MillWheel [ABB+13] which covers fault-tolerant stream processing at Internet scale. Google Cloud Dataflow combines FlumeJava and MillWheel to enable the user programming unified pipelines for batch and stream processing. Google Cloud Dataflow consists of two parts:

- A set of Software Development Kits (SDKs) to define data processing pipelines

[15] `https://github.com/cloudera/spark-dataflow`

- A google cloud platform managed service to execute the pipelines.

However, there is only a Java SDK available. Optimized Scala bindings do not exist. Since the project started in Juli 2014 and therefore is considered to be in a pre alpha status it has not been considered.

Tinkerpop Pipes

Tinkerpop[16] is a graph computing engine written in Java with multiple modules. Pipes is one out of six Tinkerpop modules. Pipes is a dataflow framework which uses process graphs. Unfortunately the framework restricts the type of exchanged objects to `Iterators` and `Iterables`. The Spark `RDD` does not implement one of them. A Workaround would be to put all `RDDs` in a `Collection`. A minor drawback is that there are no optimized Scala bindings.

4.1.2. PipE - Simple Scala Pipelining Framework

Since there is no pipelining framework available which suits our requirements, we decided to write our own lean pipelining framework called PipE. PipE claims to be a simple multi purpose linear pipelining framework. It is generic and has no dependencies especially not to Apache Spark. Thus, it is possible to use it in a completely different environment. PipE mainly consists of three classes and traits[17] which are depicted in Figure 4.1.

Building a `Pipeline` always starts with instantiating a `PipeElement` and appending or prepending other `PipeElements`. The method `append(...)` always returns the resulting `Pipeline` which can be used for execution or further appending. A `Pipeline` is just a minimalistic class to keep track of the input and output `PipeElement` and the corresponding input and output type. The `Pipeline` does not know anything about the intermediate `PipeElements`. There is no central instance which knows the structure of the `Pipeline`. Every `PipeElement` knows its neighbors and the `Pipeline` keeps track of start and end of the pipeline to provide the ability to prepend and append. The object diagram in Figure 4.2 shows an example `Pipeline` and the references between `Pipeline` and `PipeElements`.

One advantage of PipE pipelines is typesafety. It is not possible to append a `PipeElement` with an input type different to the output type of the `Pipeline`. To execute a `Pipeline` or a single `PipeElement` a method called `run(input)` is provided by the shared `Pipe` trait. It accepts input of the input type and returns output of the output type of the `Pipeline`.

[16]`https://github.com/tinkerpop`
[17]A trait is a collection of methods which can be used to extend a class. It is similar to an interface with the extension of method implementations.

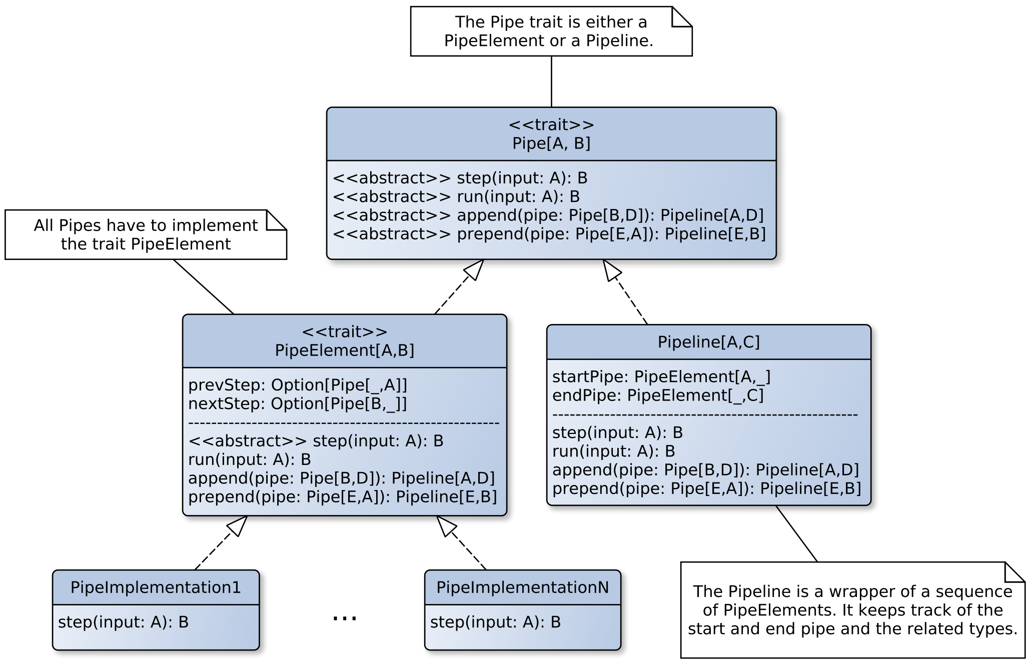

The Pipe trait is either a PipeElement or a Pipeline.
<<trait>> Pipe[A, B]
<<abstract>> step(input: A): B
<<abstract>> run(input: A): B
<<abstract>> append(pipe: Pipe[B,D]): Pipeline[A,D]
<<abstract>> prepend(pipe: Pipe[E,A]): Pipeline[E,B]
All Pipes have to implement the trait PipeElement
<<trait>> PipeElement[A,B]
prevStep: Option[Pipe[_,A]]
nextStep: Option[Pipe[B,_]]
<<abstract>> step(input: A): B
run(input: A): B
append(pipe: Pipe[B,D]): Pipeline[A,D]
prepend(pipe: Pipe[E,A]): Pipeline[E,B]
Pipeline[A,C]
startPipe: PipeElement[A,_]
endPipe: PipeElement[_,C]
step(input: A): B
run(input: A): B
append(pipe: Pipe[B,D]): Pipeline[A,D]
prepend(pipe: Pipe[E,A]): Pipeline[E,B]
PipeImplementation1
step(input: A): B
...
PipeImplementationN
step(input: A): B
The Pipeline is a wrapper of a sequence of PipeElements. It keeps track of the start and end pipe and the related types.

Figure 4.1.: Class diagram showing the three main classes of the PipE framework and its inheritance relationship. Every implementation of the PipeElement trait has to implement the method step(...) which contains the functionality of the PipeElement. The invocation of the method run(...) executes the whole pipeline.

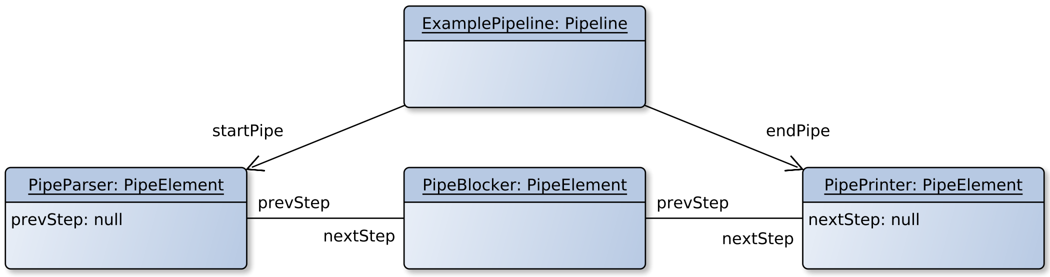
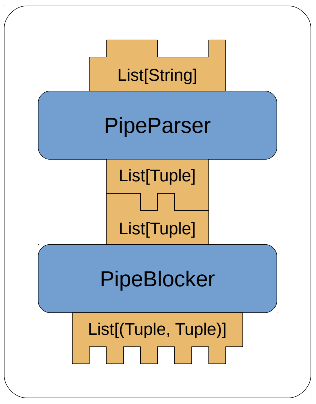
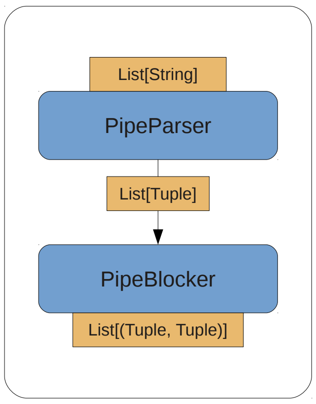


Figure 4.2.: Object diagram of an example Pipeline containing three PipeElements. The Pipeline is aware of the start and end PipeElement of the Pipeline. All PipeElements are aware of there neighbors and their input and output types.

Figure 4.3 show a simple `Pipeline` consisting of two `PipeElements`. It shows how the input and output types are visualized.



(a) Every PipeElement has an input and an output type which has to correspond to the neighboring pipe.



(b) For a better readability the in and output types are combined and an arrow symbolizes the data flow.

Figure 4.3.: Visualization of PipeElements and their corresponding in- and output types.

There is one central restriction to the creation of pipelines. PipE only allows the creation of linear pipelines. It is not possible to create branches. This is caused by the requirement of a good usability. Arbitrary graphs are hard to create and interact via a command line interface. Restricting pipelines to a linear structure seem to be the appropriate solution. To soften this limitation a `PipeContext` is introduced. A `PipeContext` is a container for arbitrary intermediate results, which can be reused by subsequent `PipeElements`. The `PipeContext` must be passed to the `run(...)` method when executing the `Pipeline`. During execution the `PipeContext` is passed implicitly from one `PipeElement` to its successor.

Implementing a `PipeElement` is simple. Listing 4.1 shows a sample implementation of a word count pipe. Its input type is an `RDD` of lines and its output type is an `RDD` of key value pairs with a word as a key and its number of occurrences as a value. There are three things regarding the creation of a simple `PipeElement`. At first, the new class has to extend `PipeElement[A, B]` with the correct input and output types `A` and `B`. The `step(input)` method has to be implemented afterwards which contains the functionality of the `PipeElement`. The third step to implement is a companion object to get a more convenient constructor syntax. If the `apply()` method is defined the `new` key word can be left out to create an object what is more convenient especially at the command line. This also unifies the creation of implemented `PipeElements` and `Pipelines`. A `Pipeline` consists of a combination of other `PipeElements` and only implements the method `apply()`, see Section 4.1.2.

```scala
class PipeWordcount()
  extends PipeElement[RDD[String], RDD[(String, Int)]] {

  def step(input: RDD[String])(implicit pipeContext: AbstractPipeContext): RDD
      [(String, Int)] = {
    // flatten the collection of word arrays
    val words = input.flatMap(line => line.split(" "))
    // initialize the counter of each word with one
    val wordsWithCounter = words.map(word => (word, 1))
    // add up all counters of the same word
    wordsWithCounter.reduceByKey(_ + _)
  }

}

// companion object for a better usability
object PipeWordcount {
  def apply() = new PipeWordcount()
}
```

Listing 4.1: Word count example Pipeline.

Last thing to know is the construction and execution of a `Pipeline`. Single `PipeElements` can be concatenated using the `append(...)` method. Every `Pipeline` or `PipeElement` can be executed by invoking the `run(...)` method. The `run(...)` method needs an `AbstractPipeContext` and the input data to be executed. Listing 4.2 shows a minimalistic example application using a parser and a blocker.

```scala
val input = Array("first line", "second line")
val pipeline = PipeParser() append PipeBlocker(param1, param2)
val pc = new DdupPipeContext()
val blockingResult = pipeline.run(input)(pc)
```

Listing 4.2: Example of Pipeline creation and execution.

Combining Pipes

To achieve a real modular architecture we need to combine pipes to create new types of pipes. For instance reading the gold standard can be split into creating id pairs from the specific input format and afterwards joining the tuples to create real tuple pairs. To achieve this all pipes are objects implementing `apply()`. This enables implementations of pipelines by only concatenating pipes and therefore make reuse of pipes possible. Listing 4.3 shows the implementation of two gold standard reader pipes. The first one reads lines in pair format and the second one reads lines in cluster format. Both make use of the `PipeReaderGoldstandardIdsToTuple` pipe.

```scala
object PipeReaderGoldstandardPairs {
  def apply(...): Pipeline[RDD[String], RDD[SymPair[Tuple]]] = {
    PipeReaderGoldstandardIdsPairs(...)
      .append(PipeReaderGoldstandardIdToTuple())
  }
}

object PipeReaderGoldstandardCluster {
  def apply(...): Pipeline[RDD[String], RDD[SymPair[Tuple]]] = {
    PipeReaderGoldstandardIdsCluster(...)
      .append(PipeReaderGoldstandardIdToTuple())
  }
}
```

Listing 4.3: Two different gold standard reader pipes implemented by combining pipes.

4.2. Architecture and Implemented Pipes

The PipE framework which is introduced in Section 4.1.2 is now applied to the duplicate detection process introduced in Chapter 3. Every single step of the process has a specific input and output type and therefore corresponds to a class of `Pipes`. In addition there are several other `Pipes` for purposes like printing statistics or writing results. All classes of `Pipes` are briefly described in the following list. To understand the following section two classes should be briefly introduced. A `SymPair` is a symmetric pair, which means `SymPair(a, b)` equals `SymPair(b, a)`. A `StringMetric[Double]` is a similarity measure to compare strings, which returns a `Double` in the range $[0, 1]$.

Reading Pipes: RDD[String] $\rightarrow$ RDD[Tuple]

The name of all pipes in the reading class are prefixed with “PipeReaderTuple”. One implementation is the `PipeReaderTupleCsv` which parses raw Comma-Separated Values (CSV) lines and creates `Tuple` instances.

Gold standard reading Pipes: RDD[String] $\rightarrow$ RDD[SymPair[Tuple]]

Like the reading pipes there are also pipes to read the gold standard. At the moment the two different formats cluster and pair are supported. The corresponding gold standard reading pipes are `PipeReaderGoldstandardCluster` and `PipeReaderGoldstandardPair`.

Preprocessing Pipes: RDD[Tuple] $\rightarrow$ RDD[Tuple]

All preprocessing pipes are prefixed by “PipePreprocessor”. They map every tuple to a new transformed version. The two main preprocessing pipes are `PipePreprocessorTrim` and `PipePreprocessorReplaceRegex`.

Indexing Pipes: RDD[Tuple] → RDD[SymPair[Tuple]]
All indexing pipes are prefixed by PipeIndexer. There are four Indexing pipes available. `PipeIndexerDummy` creates the naive search space. `PipeIndexerSortedNeighborhood`, `PipeIndexerStandard` and `PipeIndexerSuffixArray` each correspond to the equally named indexing algorithm.

Blocking Pipes: RDD[Tuple] → RDD[Seq[Tuple]]
All blocking pipes are prefixed by PipeBlocker. All three indexing algorithms except the `PipeIndexerDummy` are blocking algorithms and therefore also available as blocking pipe.

Similarity Pipes: RDD[SymPair[Tuple]] → RDD[(SymPair[Tuple], Array[Double])]
There is only one similarity pipe `PipeSimilarity` available. It takes a parameter `featureMeasures: Array[(Int, StringMetric[Double])]` which is a mapping from the feature index to the corresponding similarity measure.

Classification Pipes: RDD[(SymPair[Tuple], Array[Double])] → RDD[(SymPair[Tuple], Array[Double])]
All classification pipes are prefixed by PipeClassification. They are split into two groups trained and untrained classification pipes. The trained ones are three machine learning algorithms. Available are `PipeClassificationNaiveBayes`, `PipeClassificationDecisionTree` and `PipeClassificationSvm`. The untrained classifier available is the `PipeClassificationThreshold`, which is a threshold-based classifier. The `PipeClassificationThreshold` is detailed in Subsection 4.3.3.

Clustering Pipes: RDD[(SymPair[Tuple], Array[Double])] → RDD[Set[Tuple]]
All clustering pipes are prefixed with `PipeClustering`. At present, there is only one clustering pipe `PipeClusteringTransitiveClosure` to compute the transitive closure. To implement new clustering algorithms there is a base class `AbstractPipeClusteringGraph` available which uses GraphX to create a graph out of the classification result.

Passthrough Pipes: A → A
All other pipes are `PassthroughPipes[A]`. A `PassthroughPipe[A]` is a generic pipe which does not modify the input data. All `PassthroughPipes` have the same input and output type. There are Analyzing Pipes which analyze the input data. There are printing Pipes which for instance print out a sample of 10 elements of the input data. Optimization Pipes are pipes to optimize the performance. They are used to cache or persist RDDs. Writing pipes are intended to write results to the hard disk in a special format.

4.3. Usage of the Framework

The SddF framework is designed for a convenient usage. In general, the execution of a duplicate detection pipeline comprises three steps. These steps are depicted in Figure 4.4.

1. Define and execute corpus pipeline
2. Define and execute gold standard pipeline
3. Define and execute duplicate detection pipeline

The following three sections address these three steps by means of an example.

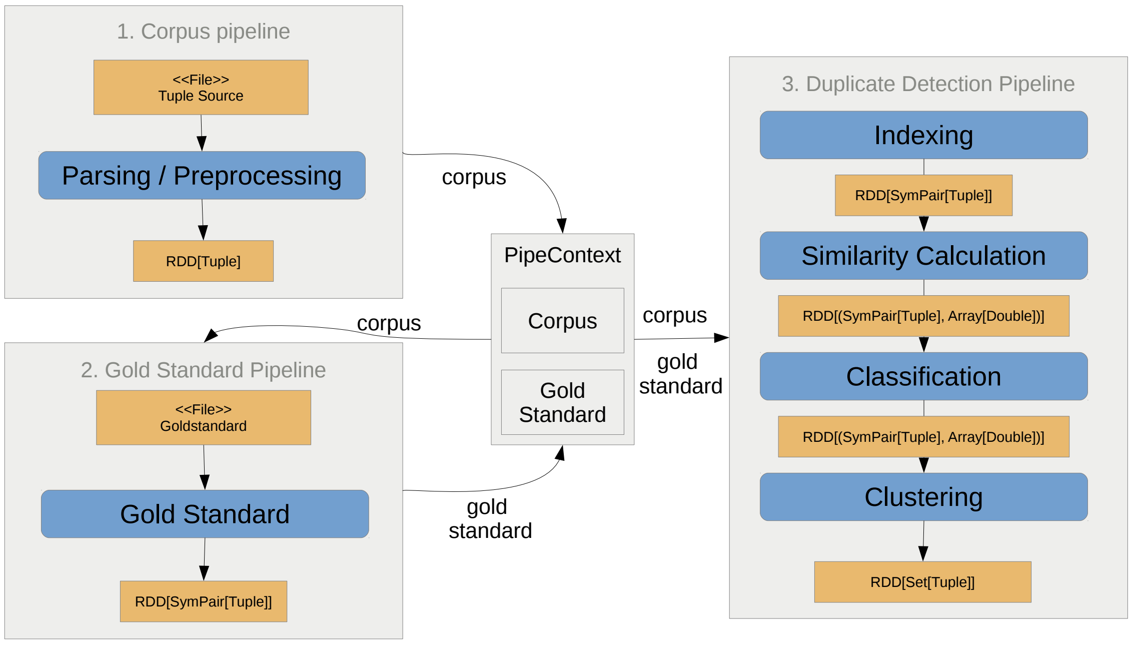


Figure 4.4.: The Implementation of the duplicate detection is separated into three pipelines which are connected by the PipeContext.

4.3.1. Corpus Pipeline

The result of the corpus pipeline is a set of tuples. The main data structure of the `Tuple` class to store the features in is an `Array[String]`. An `Array` was preferred over a `Map` because it is more space efficient. The drawback of an `array` is the indexed feature access. To access the features in a more convenient way every feature needs a corresponding index name pair.

```
val Number = (0, "number")
val Title = (1, "title")
val Length = (2, "length")
val Artist = (3, "artist")
val Album = (4, "album")
val Year = (5, "year")
val Language = (6, "language")
```

Out of all index name pairs a feature index name mapping is created. This mapping is used to translate an index to a name and vice versa. For instance it is used to include names instead of numbers in the debug output.

```
val featureIdNameMapper = new FeatureIdNameMapping(Map(Number, Title, Length,
    Artist, Album, Year, Language))
```

The next step is the definition of the corpus pipeline. In this case we define the input pipeline the following way.

```
val allFields: Seq[Int] = Seq(Number, Title, Length, Artist, Album, Year,
    Language)
val allFieldsWithId: Seq[Int] = Ignore +: Id +: Ignore +: allFields

var inputPipeline = PipeReaderOmitHead()
  .append(PipeReaderTupleCsv(allFieldsWithId))
  .append(PipePreprocessorTrim(allFields: _*))
  .append(PipePreprocessorRemoveRegex("[^0-9]", Number, Year, Length))
  .append(PipeAnalyseCorpus())
  .append(PipeStoreInContextCorpus())
  .append(PipeOptimizePersistAndName("corpus"))
```

`PipeReaderOmitHead` skips the column name header. The `PipeReaderTupleCsv` creates a tuple for each line of the input CSV file. The parameter `Sequence[Int]` is used to assign each feature to a column of the CSV file. Columns can also be ignored by the `Ignore` keyword and one column must be marked to be the tuple id with the keyword `Id`. In our example the first column of the input file is ignored, the second one is the id column, the third one is ignored and all other columns are continuously assigned to the feature array. It will be converted to a numeric id instead of being stored as a `String` like all other features. After the reading step all features are trimmed and the integer features are cleaned from all characters except digits. The `PipeAnalyseCorpus` analyzes the size of the corpus (i.e. the number of tuples). The next pipe stores the corpus in the pipe context to make it accessible to all succeeding pipes. The execution of the input pipeline results in an `RDD[Tuple]`.

```
val pc = new SddfPipeContext("preRunPipe")
val rawData: RDD[String] = sc.textFile("inputfile.csv")
val corpus: RDD[Tuple] = inputPipeline.run(rawData)(pc)
```

4.3.2. Gold Standard Pipeline

The gold standard is needed to evaluate the results of the duplicate detection pipeline and to train a classifier like a decision tree. If an untrained classifier or previously created model is loaded and the analyzing is skipped the gold standard is not needed. The gold standard can be read from pair or cluster format. The pair format is a tuple id pair in each line. The cluster format assigns a cluster id to each tuple id in a single line. In this example we use the cluster format.

```
val goldstandardPipe = PipeReaderOmitHead()
    .append(PipeReaderGoldstandardCluster())
    .append(PipeStoreInContextGoldstandard())
    .append(PipeOptimizePersistAndName("goldstandard"))
```

The execution of the gold standard pipeline is similar to the input pipe. The same `SddfPipeContext` object is referenced.

```
val gsRawData: RDD[String] = sc.textFile("goldstandardfile.csv")
val goldstandard: RDD[SymPair[Tuple]] = goldstandardPipe.run(gsRawData)(pc)
```

4.3.3. Duplicate Detection Pipeline

After reading and creating the corpus and the gold standard we can start to define our duplicate detection pipeline. First step is to instantiate an indexing pipeline. For this example we choose a `PipeIndexerStandard` which is a standard blocker. The `BlockingKeyBuilderBasic` creates a blocking key for each tuple. It takes a sequence of pairs of a feature index and a `Range` indicating the characters which will be added to the blocking key. The following blocking key builder concatenates the first 10 characters of the Title, Artist and Album feature. The `PipeAnalyseIndexer` prints out some statistics about the blocking result. The `PipeOptimizePersistAndName` pipe explicitly caches the blocking result in memory.

```
val bkvBuilder = new BlockingKeyBuilderBasic(
  (Title, 0 to 10), (Artist, 0 to 10), (Album, 0 to 10)
)

val indexPipeline = PipeIndexerStandard(bkvBuilder)
  .append(PipeAnalyseIndexer())
  .append(PipeOptimizePersistAndName("blocked-pairs"))
```

Next step is to create a similarity pipeline, which calculates a similarity vector for each tuple pair in the search space. To do this, we have to configure a measure for each feature. In our example we only use two different measures. The `JaccardMetric(2)` computes the Jaccard Index of the sets of bigrams of both strings. The `MeasureEquality` is a binary measure which results in 1 if both features are equal and otherwise result in 0.

```
val featureMeasures: Array[(FeatureId, Measure)] = Array(
  (Number, MeasureEquality)
  , (Title, JaccardMetric(2))
  , (Length, MeasureEquality)
  , (Artist, JaccardMetric(2))
  , (Album, JaccardMetric(2))
  , (Year, MeasureEquality)
)

val similarityPipeline = PipeSimilarity(featureMeasures)
```

The result of the similarity pipeline is an `RDD[(SymPair[Tuple], Array[Double])]` which is a pair consisting of a tuple pair and a similarity vector. To decide whether a pair is a match or a non-match, a classification pipeline is needed. It will filter out all non-matches. In this case an untrained threshold classifier is used. To configure the `PipeClassificationThreshold` a threshold must be assigned to each feature. If a similarity vector exceeds all thresholds it will be labeled as match.

```
val thresholds: Array[(FeatureId, Threshold)] = featureMeasures.map(pair => {
  (pair._1, 0.8)
})

val classificationPipeline = PipeClassificationThreshold(thresholds)
  .append(PipeAnalyseClassification())
  .append(PipeOptimizePersistAndName("duplicate-pairs"))
```

Last step in the pipeline is the clustering step. In this case the transitive closure is computed and the result gets analyzed.

```
val clusteringPipeline = PipeClusteringTransitiveClosure()
  .append(PipeAnalyseClustering())
}
```

Putting together all four pipelines results in one duplicate detection pipeline. It is executed by invoking the run method and passing the corpus and a pipe context.

```
val erPipeline = indexPipeline
  .append(similarityPipeline)
  .append(classificationPipeline)
  .append(clusteringPipeline)
```

```
val entityClusters: RDD[Set[Tuple]] = erPipeline.run(corpus)(pc)
```

Executing the duplicate detection pipeline results in an `RDD[Set[Tuple]]`. All tuples in a set are supposed to be duplicates and correspond to one entity.

4.3.4. Exploratory / Interactive Usage

When duplicate detection and big data come together, long runtimes can be expected. To tackle these long runtimes a distributed batch processing can be done, but the duplicate detection process can be highly parameterized. The optimal choice of these parameters depends on the size, shape and structure of the data. For that reason, the process needs to be individually parameterized for every data source. This optimization process is very time consuming especially in case of a batch system. An automated parameter optimization like a grid search can be applied, but in general it is not possible to automate the creation of the whole duplicate detection process. It is still a manual process, which would be more intuitive to optimize in an exploratory way than repeating the execution of batch processes. Exploratory means interactively executing parts of the process and analyzing intermediate results. Exploratory means especially to provide a flexible and open API and to let the user decide how he wants to use it to explore the data. The term was chosen in dependence on the concept of Exploratory Data Analysis (EDA) [Tuk93] introduced by John Tukey in 1977.
We try to combine cluster computing with an exploratory duplicate detection process parametrization. The motivation for the exploratory approach is to enable developers and researchers to rapidly prototype their duplicate detection process and to enable them to flexibly explore insights of the process.
That is where the interactive spark shell comes into play. After starting the spark shell pipelines can be created, connected, executed, analyzed and compared interactively in the shell.
A small example outlines the interactive usage. We need to start the spark shell with the SddF jar loaded. This is done by referencing it with the --jars option. To avoid typing always the same boiler plate imports and code it is possible to execute Scala scripts via the -i parameter during the shell startup. The ShellPrerequesites.scala imports all public `Pipes` to make them accessible in the shell. The second script MusicbrainzTemplate.scash contains the input pipeline which creates the corpus. Starting the spark shell with these options results in the following prompt. To improve the readability the output has been shortened.

```
niklas@niklas-t420s:~$ ~/opt/spark-1.3.1-bin-hadoop2.6/bin/spark-shell --jars
    target/scala-2.10/SddF-assembly-0.1.0.jar -i src/main/scala/de/unihamburg/
    vsis/sddf/shell/ShellPrerequesites.scala -i src/main/scala/de/unihamburg/
    vsis/sddf/shell/MusicbrainzTemplate.scash
Welcome to
      ____            __
     / __/__ ___ _____/ /__
    _\ \/ _ \/ _ ‘/ __/ ’_/
   /___/ .__/\_,_/_/ /_/\_\ version 1.3.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_79)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
scala>
```

Target of this small example is to compare the block size distribution of Standard Blocking and Suffix Array Blocking. To do this at first we have to create both blocking results.

```
scala> val blocksStandard = PipeBlockerStandard(bkvBuilder).run(corpus)
blocksStandard: RDD[Seq[Tuple]]
scala> val blocksSuffixArray = PipeBlockerSuffixArray().run(corpus)
blocksSuffixArray: RDD[Seq[Tuple]]
```

Since the result type is an `RDD` the Spark API can be used. To get the distribution each block is mapped to a key value pair. The key is the size of the block and the value is a constant 1. Afterwards the `reduceByKey(_ + _)` sums up all values of the same block size.

```
scala> val blockSizeDisitributionStandard = blocksStandard.map(x => (x.size, 1)
    ).reduceByKey(_ + _)
blockSizeDisitributionStandard: RDD[(Int, Int)]
scala> val blockSizeDisitributionSuffixArray = blocksSuffixArray.map(x => (x.
    size, 1)).reduceByKey(_ + _)
blockSizeDisitributionSuffixArray: RDD[(Int, Int)]
```

To print out the resulting distribution we need to transfer them to the driver by calling `collect()`. This creates a Scala collection which can be sorted by calling `sorted()`. Last thing to do is to iterate over all elements and print them out.

```
scala> blockSizeDisitributionStandard.collect().sorted().foreach(println(_))
(2,433)
(3,150)
(4,67)

scala> blockSizeDisitributionSuffixArray.collect().sorted().foreach(println(_))
(2,12916)
(3,4314)
(4,1874)
(5,188)
(6,46)
(7,44)
(8,19)
(9,35)
(10,1)
(11,3)
(12,10)
```

The result looks as expected. The Standard Blocker produces fewer and smaller blocks since the blocking keys are more special. This is only a very short introduction to the spark shell using the SddF framework. To explore intermediate results of the deduplication process the full Spark and Scala API can be used.
To improve the usage of the command line and the lookup of pipes a naming scheme is applied to all pipe names. Subclasses of `PipeElement` which are supposed to be instantiated should follow this naming scheme. The class name should start with the prefix "Pipe" followed by a number of grouping key words. Each grouping key word starts with an upper case letter. The order of the grouping key words is from general to special in order to create a grouping of `PipeElements` which is usable through the autocompletion of the spark shell. The following Example should clarify the naming scheme.
Assuming we have two different parser pipes. The first one parses files in CSV format. The second one parses files in a custom binary format. A sensible naming according to the naming scheme would be "PipeParserCsv" and "PipeParserBinary". To search a parser on the command line the only thing to do is typing "PipeParser" and hit <tab> and all classes in the classpath sharing this prefix will be listed.

5. Evaluation

In this chapter we evaluate the implementation described in Chapter 4 with respect to time and space. Section 5.1 explains the creation of the input data sets. Afterwards the cluster setups are described in Section 5.2. Section 5.3 is about the experiments.

5.1. Data Sets

A problem is to acquire large structured data sets which are suited for duplicate detection purposes. Our approach is to get one large base set which is supposed to contain no duplicates. This base set is used to derive input sets which are used in the experiments. Deriving in this case means selecting a number of tuples from the base set, inserting duplicates and applying error generators on all tuples.

5.1.1. Base Set

As base set we extracted $1.6 \cdot 10^7$ audio track tuples from the open “Musicbrainz” PostgreSQL database[18]. The “Musicbrainz” database aims to build a database with all relevant information about music releases. A working Virtual Machine (VM) of the Musicbrainz database server, which has a size of about 11 GB, can be obtained on the Musicbrainz website[19]. The schema of the database is complex. It is depicted in Appendix A.1. We extracted the tuples using a huge join shown in Listing 5.1. The base set evolved from a cooperation with Kai Hildebrandt, who uses the same data set in his master's thesis [Hil15].

```
set client_encoding='utf8';
\copy (
  SELECT tr.id, tr.number, tr.name AS title, tr.length, ac.name
  AS artist, rl.name AS album, rc.date_year AS year, lg.name AS language
  FROM track AS tr
  LEFT JOIN artist_credit AS ac ON tr.artist_credit = ac.id
  LEFT JOIN medium AS md ON tr.medium = md.id
  LEFT JOIN release AS rl ON md.release = rl.id
```

[18]https://musicbrainz.org
[19]https://musicbrainz.org/doc/MusicBrainz_Server/Setup

```
  LEFT JOIN release_country AS rc ON rl.id = rc.release
  LEFT JOIN language AS lg ON rl.language = lg.id
  WHERE ac.name != 'Various Artists'
) To '/tmp/musicbrainz-raw.csv' With CSV HEADER;
```

Listing 5.1: Join operation used to extract the audio track tuples from the Musicbrainz database.

After the extraction of the data the tuple set contains several duplicates. To remove all duplicates from the file we used the bash script in Listing 5.2. The output of the script is the base input set.

```
#!/bin/sh
sort -k2 -t, -u $1 > $1.unique.sorted
sort --random-sort $1.unique.sorted > $1.unique.random
```

Listing 5.2: Script to remove duplicate lines from a textfile.

If the amount of input data is not sufficient it can be artificially expanded. Simple duplication of the data set would not work since the mean duplicate cluster size expands linearly which is not wanted. Vernica et al. proposed an algorithm [VCL10, p. 8] which increases the corpus and linearly increases the gold standard size without increasing the mean cluster size. For our purpose an input set size of 10^7 is sufficient, but to enlarge the input data set this algorithm is promising.

5.1.2. Derived Sets

To insert duplicates and apply errors on the tuples we used a software called “Dapo” which emerged from the master’s thesis of Kai Hildebrandt [Hil15]. We created twelve different input sets. The final input set sizes are 10^4, 10^5, 10^6 and 10^7 tuples. Each set is populated with either 1%, 10%, or 50% duplicates. The duplicate cluster distribution is fixed. The fraction of duplicate tuples in the input data set is later on referred to as dirtiness. A short example shall outline the procedure. We want to create the file “10_000-0.5.csv”, which means $10,000$ tuples in the result file and 50% duplicate tuples in the result file. When creating the file, the first $5,000$ tuples are taken from the base set. Afterwards $5,000$ duplicate tuples are inserted according to the duplicate cluster size distribution. All cluster sizes are distributed in that way to produce the same number of tuple pairs. That means the tuple pairs produced by each cluster size are equally distributed. In this case, we use duplicate cluster sizes from $2-5$. The distribution can be seen in Figure 5.1. The result is a file with $10,000$ tuples and a dirtiness of 0.5.
After inserting the duplicates a probabilistic error generator is applied to the whole set. There are two categories of errors, weak errors and hard errors. Weak errors are a typo,

an Optical Character Recognition (OCR) error, an encoding error, a conversion to American Standard Code for Information Interchange (ASCII), an insertion of random blanks and a removal of single digits. Hard errors are removing or abbreviating a feature and transposing two features. The chance that a tuple is selected for the application of a weak error is 50%. If the tuple is selected, the chance that a feature of this tuple is selected for the application of a weak error is 30%. After the feature selection every weak error is applied in a sequence with a probability of 20% to all of the selected features. Therefore, multiple applications of different weak errors on the same feature are possible.
The probability that a tuple is selected for the application of a hard error is 20%. The probability that a feature is selected for the application of a hard error is 15%. At first the weak errors are applied to the data and afterwards the hard errors. Therefore, a hard error may overwrite a weak error. An overview of the derived data sets is depicted in Table 5.1.

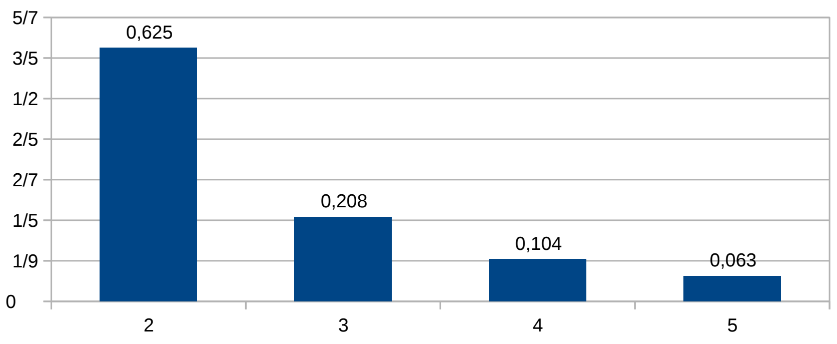


Figure 5.1.: Cluster size distribution of the inserted duplicate clusters.

Name	Size (MB)	# Tuples	# Duplicates
10_000-0.01.csv	∼ 1	10 000	100
10_000-0.1.csv	∼ 1	10 000	1000
10_000-0.5.csv	∼ 1	10 000	5000
100_000-0.01.csv	∼ 10	100 000	1000
100_000-0.1.csv	∼ 10	100 000	10 000
100_000-0.5.csv	∼ 10	100 000	50 000
1_000_000-0.01.csv	∼ 100	1 000 000	10 000
1_000_000-0.1.csv	∼ 100	1 000 000	100 000
1_000_000-0.5.csv	∼ 100	1 000 000	500 000
10_000_000-0.01.csv	∼ 1000	10 000 000	100 000
10_000_000-0.1.csv	∼ 1000	10 000 000	1 000 000
10_000_000-0.5.csv	∼ 1000	10 000 000	5 000 000

Table 5.1.: Overview of all derived data sets. The number of duplicates are contained in the number of tuples.

5.2. Cluster Setup

5.2.1. Cluster Hardware

The hardware used to build the cluster are standard desktop computers. The hardware specification is shown in Table 5.2.
The network topology is a switched star with full duplex realized by a 1 Gbit/s Switch. In the different cluster setups all nodes are running on machines of the same hardware configuration.

Quantity	Type	CPU	RAM	Network
5	Dell Optiplex 980	4×2.67 GHz (i5 750)	8 GB	1 Gbit/s

Table 5.2.: Overview of the cluster node hardware.

5.2.2. Cluster Configurations

All cluster configurations used in the experiments consist of one master node and n worker nodes. When referring to the size of the cluster, it is always the number of worker nodes. The different cluster configurations are listed in Table 5.3. The default cluster configuration is 4-worker.

Name	# Worker Nodes	RAM (GB)	Cores	Network
1-worker	1	6	4	1 Gbit/s
2-worker	2	12	8	1 Gbit/s
3-worker	3	24	12	1 Gbit/s
4-worker	4	32	16	1 Gbit/s
local	0	8	4	-

Table 5.3.: Overview of the different cluster configurations.

5.2.3. Cluster Software

On the master and all worker nodes an Ubuntu Server 12.04 is installed. No virtualization layer is used. The Hadoop version is 2.6.0 and Apache Spark version is 1.3.1. Spark is configured to run in standalone mode. Hadoop YARN or other cluster manager are not used. Only HDFS is used from the Hadoop Stack to have a convenient distributed file access. HDFS is configured to store files fully redundant. That means every file is present on every worker node. The master node is dedicated to run the HDFS NameNode and the Spark Master. Every worker node runs a HDFS DataNode and a Spark Worker. During the experiments the HDFS cluster is always congruent with the Spark cluster. That means a file in the HDFS should never be accessed via network because they are always accessible locally.

5.3. Experiments

The experiments focus on the scalability of the duplicate detection workflow with respect to the amount and shape of input data and the number of worker nodes. The crucial step is the choice of the indexing algorithm because it determines the size of the resulting search space. There are three different indexing algorithms implemented: Standard Blocking [FS69, p. 64], Sorted Neighborhood [HS95, HS98] and Suffix Array Blocking [AO05]. In addition to the algorithms, a multi blocker is implemented to coalesce the result of multiple blocking algorithms. Unfortunately, the Sorted Neighborhood always crashed with an `ArrayIndexOutOfBoundsException` while processing one of the largest input sets. The exception occurs during the clustering phase. It seems to be related to an internal Spark bug "SPARK-5480"[20] of GraphX. The bug could not be isolated. For that reason, the Sorted Neighborhood Blocking is skipped completely during the experiments. The performance of the algorithms is highly dependent on several parameters. The launch of the experiments is mostly automated by Python scripts. The script launches the cluster with the desired number of worker nodes and executes the configured pipelines. This process is repeated automatically with all numbers of worker nodes. Running the experiments is still very time consuming since the amount of data is quite large for a small number of worker nodes. The runtime of a single pass of a given input set and a pipeline can take up to hours depending on the cluster and pipeline configuration. For that reason only a selected range of parameter combinations is analyzed and plotted.
Visualization of the result is also not trivial since it is a 4 dimensional space which would yield in a hyper plane. The four dimensions are the three input dimensions number of worker nodes, dirtiness and input set size as well as the result dimension runtime. To visualize the result the dimension of worker nodes is reduced by plotting multiple lines. The resulting three dimensional space is sliced by a fixed dirtiness or fixed input set size to get a two dimensional visualization.
The resulting runtime always refers to the overall runtime of the whole pipeline. Start of the pipeline is loading the input data and end of the pipeline is the analytics of the clustering. The result, neither intermediate nor final, is explicitly written to disk.

[20]`https://issues.apache.org/jira/browse/SPARK-5480`

5.3.1. General Pipeline Setup

In general, the following configuration of the duplicate detection pipeline was chosen. The tuple, initially parsed from the input data, consists of 7 features and an unique id.

```
Schema: (id, track-number, title, length, artist, album, year, language)
```

To ensure type and data quality aspects some preprocessing steps where applied afterwards. A trim operation is applied to all fields and on the integer fields number, year and length all characters except numbers are removed. The corpus which is the tuple set after preprocessing is explicitly cached by Spark in memory because it is used in succeeding steps in the pipeline.
The gold standard is read from the same input file which contains the cluster ids of all tuples. Also the gold standard is stored as a set of tuple pairs. It is also explicitly cached by Spark for later evaluation purposes.
The blocking key is a concatenation of the first 2 characters of the fields title, artist and album resulting in a key with a maximum length of 6.

```
new BlockingKeyBuilderBasic((Title, 0 to 2), (Artist, 0 to 2), (Album, 0 to 2))
```

The blocking algorithm chosen is Standard Blocking, since it is a widely known and utilized algorithm. In the plots the blocker described is named "Standard-Blocker-3x2", because the blocking key consists of three parts of a length of two. The notation like "3x2" always refers to the blocking key. The first number is the number of features incorporated, which is always three, and the second number is the length of the prefix extracted from a feature. The result of the blocking is explicitly cached by Spark to avoid expensive recalculation of the search space. The similarity calculation is done on all fields except the language, because it turned out that this feature is very sparse. For the numeric features number, length and year an equality similarity measure was chosen. The equality measure is a binary similarity measure which returns 1 on equality and 0 else. All other fields are compared after being converted to lower case with the Jaccard Metric [NH10, p. 24] with a q-gram size of two.
The classification step is done by a threshold based classifier. If all dimensions of the similarity vector exceed the threshold of 0.8, the tuples are labeled as duplicates. Afterwards, the transitive closure is computed which is the result of the duplicate detection process. The last step is the evaluation of the result which includes recall, precision, number of clusters and average cluster size. Moreover, the intermediate result after each step is analyzed. This pipeline configuration is referred to as standard pipeline.

5.3.2. Measurements

In the focus of the measurements are time and space. How does the variation of the number of worker nodes, the input set size and the number of duplicates influence the runtime and space needed by the application? The runtime of the duplicate detection pipeline described in Section 5.3.1 is analyzed while varying the input set size, the number of duplicates and the size of the cluster. Moreover, the intermediate result size of the pipeline is analyzed. The cluster configuration used is 1-worker, 2-worker, 3-worker and 4-worker.
Figure 5.2 shows the runtime while varying the input set size. The process scales with respect to the worker nodes and input set sizes. Figure 5.3 shows the slice of Figure 5.2 at 10^6 varying the dirtiness. Varying the dirtiness does not have the same impact than varying the input set size. The process scales very well on different dirtiness setups. Only during the processing of the input set size of 10^6 on a single node the runtime increases more than linear. The reason might be that the application runs out of memory and has to spill data to the disk or has to drop more RDD partitions which later have to be recomputed. The increase of 3 to 4 worker nodes does hardly result in a lower runtime. It would be interesting to do the same experiments on 5 and 6 worker nodes to investigate this tendency. In general, when processing large numbers of input data or input data with a high dirtiness, the increased runtime can be countered by increasing the number of cluster nodes.
Figure 5.4 shows the sizes of the intermediate results of the duplicate detection pipeline. The four intermediate results are the corpus size, the search space size, the classification result set and the final clustering result. The corpus size is the number of input tuples. The search space size and the number of matches are the quantity of tuple pairs. The number of duplicate clusters are not the duplicate pairs, but the quantity of clusters. The experiment verifies a simple conclusion. Increasing the dirtiness means increasing all intermediate result sizes. Raising the dirtiness from 0.01 to 0.5 will increase the final result size of two orders of magnitude resulting in a longer runtime and higher memory consumption. Another conclusion is the larger the input set size, the less relevant is the dirtiness for the search space size. On the small input set size of 10^4 the search space size varies from 184 (0.01) to 5309 (0.5) what is a ratio of 28.9. On the large input set size of 10^7 the search space size varies from $1,682,734$ (0.01) to $2,187,763$ (0.5) what is a ratio of 1.3.
Also trivial, but worth mentioning, is the decrease of the efficiency of standard blocking on larger input sets. All blocking algorithms have this problem since the average block size increases with larger input set sizes. The number of pairs have a quadratic complexity depending on the block size. A solution to solve blocking problems on large input set sizes is multi blocking. Multi blocking incorporates multiple blocking algorithms and combines the results. To reduce the search space size an intersection of all blocking results would

be a possible solution. The upper bound for the maximum possible recall is given by the smallest recall of all blocking algorithms incorporated.

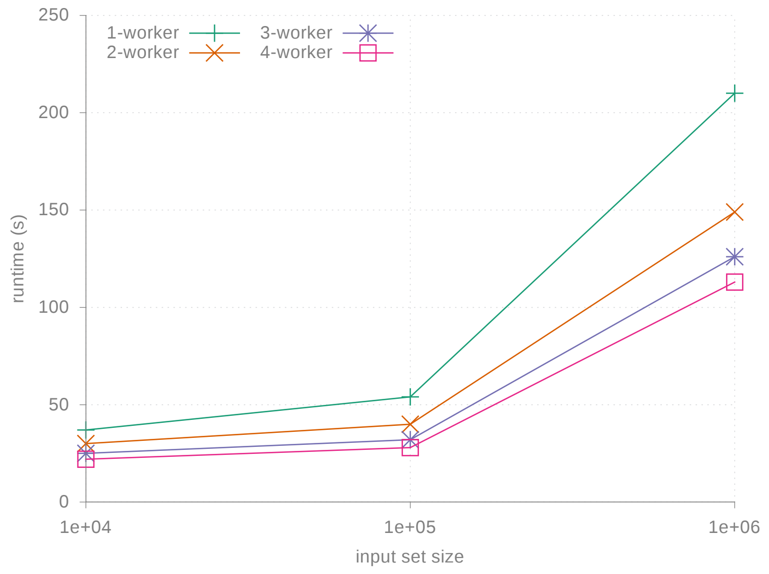


Figure 5.2.: Runtimes of the duplicate detection pipeline with a fixed dirtiness of the data of 0.1 and a variable corpus size.

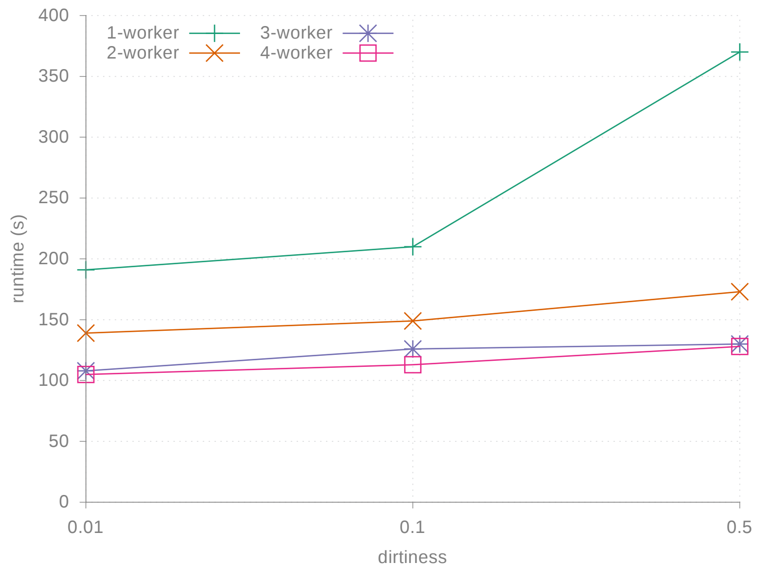


Figure 5.3.: Runtimes of the duplicate detection pipeline with a fixed corpus size of 10^6 and a variable dirtiness of the data.

5.3.3. Varying Indexing Algorithms

The indexing algorithm is the crucial part of the duplicate detection pipeline, which determines runtime and space needed. Standard Blocking, with an appropriate blocking

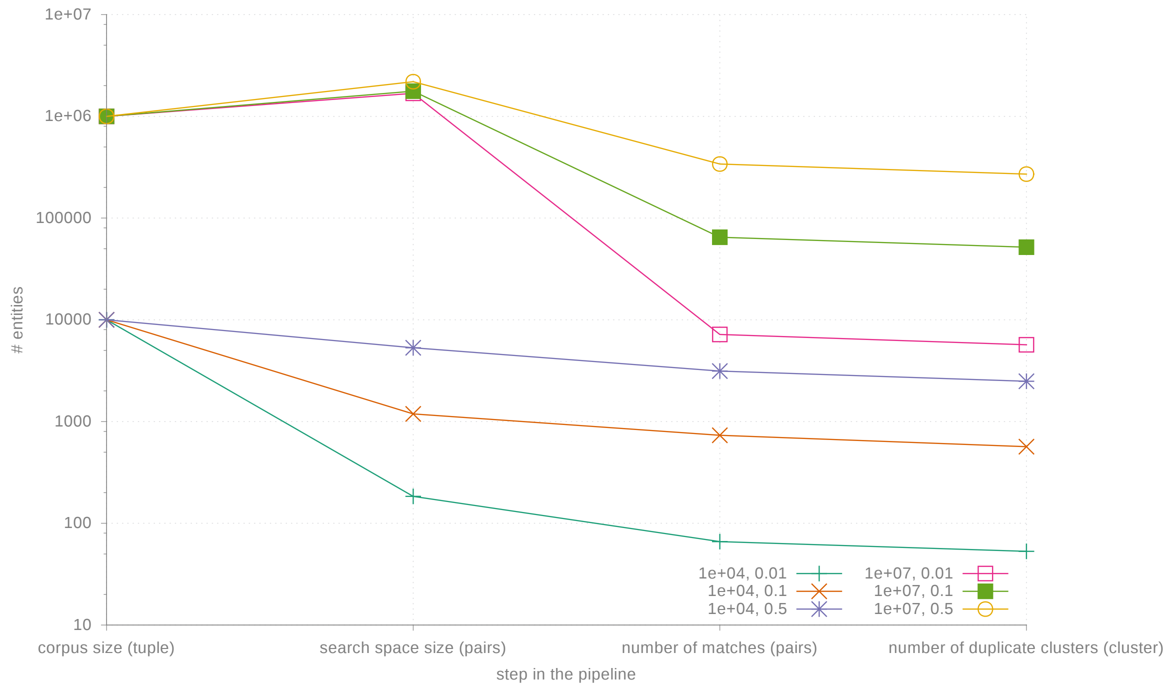


Figure 5.4.: The Figure shows the sizes of the intermediate results of the pipeline for the smallest and largest corpus. The left most value is the number of tuples in the corpus. All other values are tuple pairs. Remind the logarithmic y axis.

key, can be space and time efficient, but tends to exclude many duplicate pairs from the search space, decreasing the recall. For that reason, another established blocking algorithm called Suffix Array Blocking is analyzed in this section. Suffix Array Blocking has two Parameters, maximum block size and minimum suffix length, as described in Section 3.2.3. Increasing the maximum block size leads to large blocks sharing a common suffix. Decreasing it leads to small blocks with uncommon suffixes. Increasing the minimum suffix length leads to smaller blocks with uncommon suffixes. Decreasing it leads to larger blocks with common suffixes. Both parameters are tightly coupled. Increasing the minimum suffix length will not necessarily have an impact if the maximum block size is too small. All enlarged blocks would be discarded by the maximum block size threshold. For our comparison a Suffix Array Blocker with a minimum suffix length of 8 and a maximum block size of 18 is incorporated. The blocking key chosen is the concatenation of the prefixes of the three features title, artist and album of a length of 5 characters.

Figure 5.5 compares the runtime of the whole pipeline using a Standard Blocker and a Suffix Array Blocker. The Suffix Array Blocker scales better with respect to the number of worker nodes. The difference in space consumption can be seen in Figure 5.6. The search space of Standard Blocking is larger than the one of Suffix Array Blocking. The quality of the results is depicted in Figure 5.7 for the Standard Blocker and in Figure 5.8 for the Suffix Array Blocker. The measures are recall and precision of the final result and rss-recall and rss-precision. Rss stands for reduced search space, which is the result of the indexing step. The rss-recall shows the maximum recall that can be achieved by the whole process. The rss-precision shows the fraction of duplicate pairs in the search space. All measures are nearly constant except the rss-precision. It decreases with increasing input set size. Thus, the fraction of non duplicate pairs in the search space increases with enlarging the input set. An explanation could be that the average block size increases with enlarging the input set. Since the resulting number of pairs in the search space is a quadratic function of the block-size, that would explain the decrease of the rss-precision.

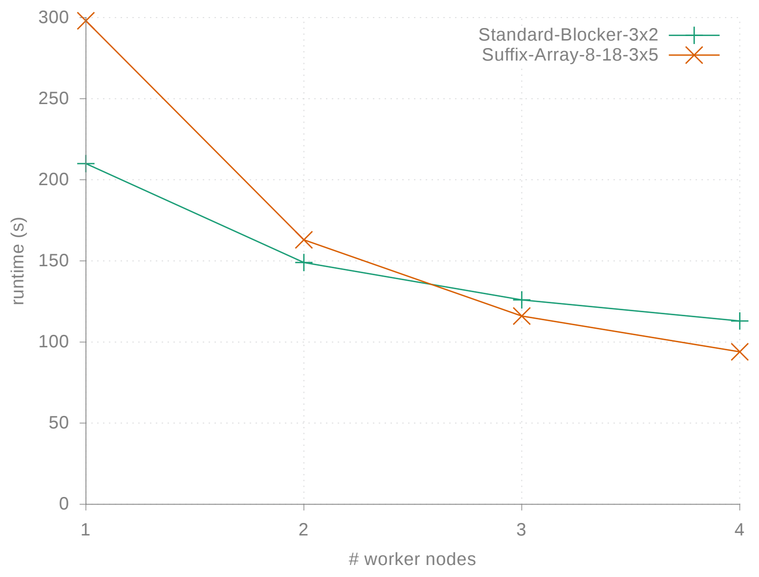


Figure 5.5.: Comparison of the runtime of the two blocking algorithms on an input set size of 10^6 and a dirtiness of the data of 0.1.

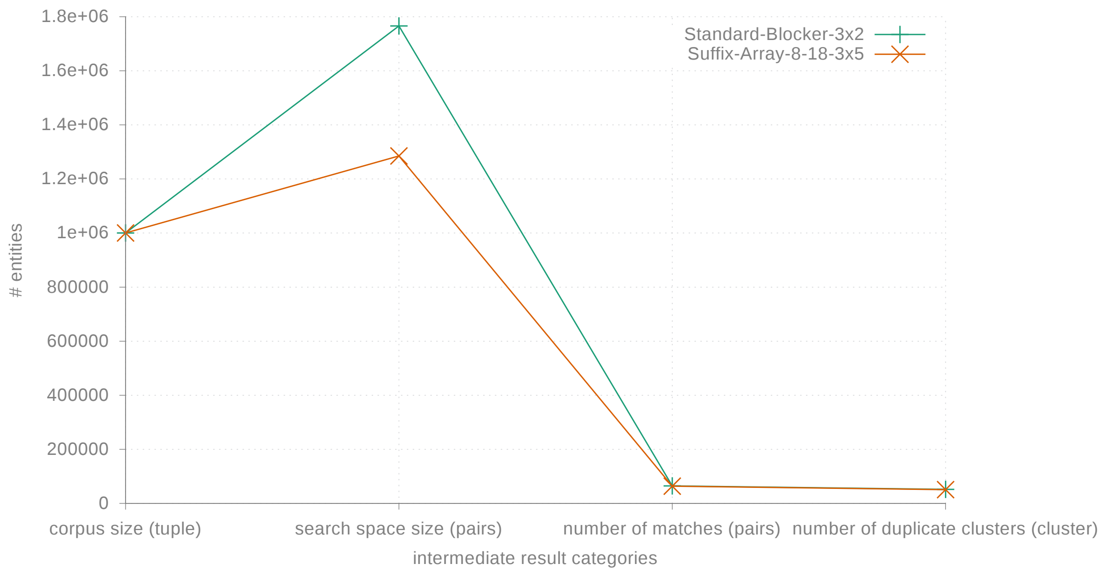


Figure 5.6.: Comparison of the space needed by the two blocking algorithms on an input set size of 10^6 and a dirtiness of the data of 0.1.

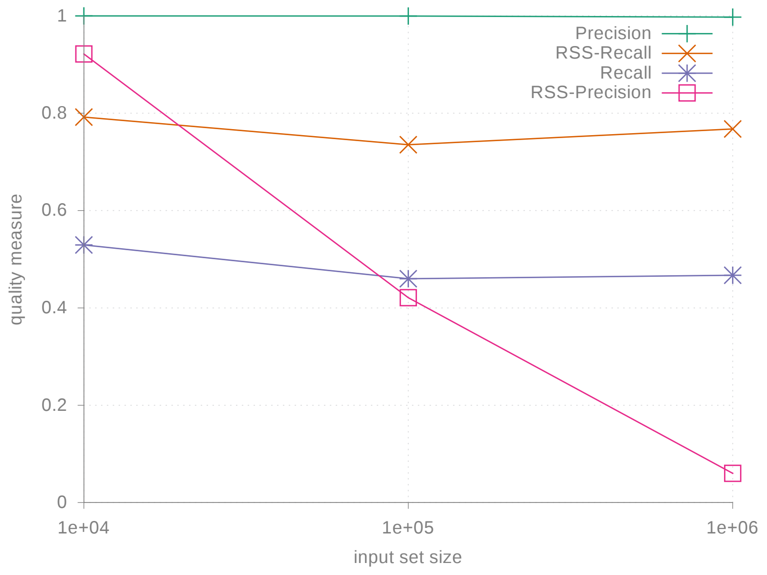


Figure 5.7.: Quality measures of the run of the Standard Blocker.

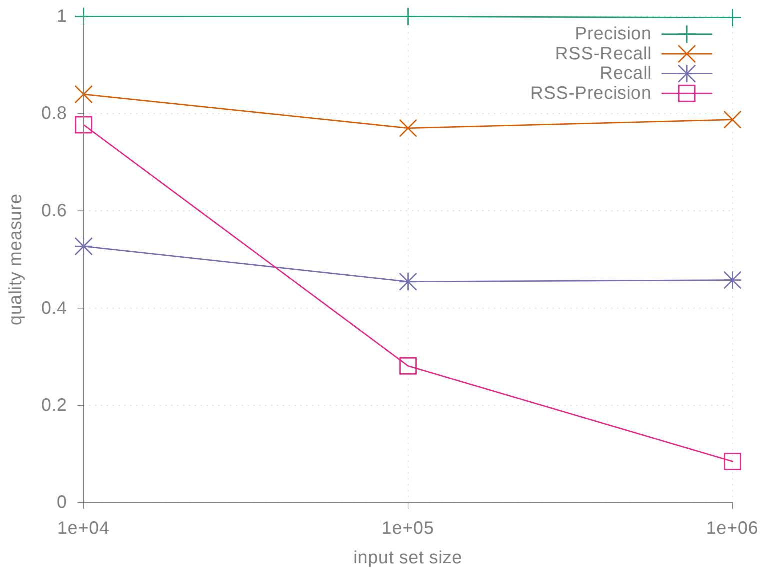


Figure 5.8.: Quality measures of the run of the Suffix Array Blocker.

5.3.4. On an input set size of 10^7

Since the input set of a size of 10^7 is too large to be processed by the smaller cluster configurations, it is only processed by 4-worker. Two different Blocker, Suffix Array and Standard Blocker, are analyzed. The pipelines are configured as in Section 5.3.3 and analyzing is disabled. Figure 5.9 shows the runtime of both pipelines on all file sizes. On large file sizes the Suffix Array Blocker is faster than the Standard Blocker. The increase of the runtime from the input set size of 10^6 to 10^7 is remarkable. Figure 5.10 shows the intermediate result sizes of the Standard Blocker for the two largest input set sizes. It shows the reason why the runtime increases that much. The search space of the input set size of 10^7 has a size of 1.6^8, which is about 100 times larger than the one of the input set size of 10^6. The runtime of the Suffix Array Blocker is 22 min 25 s (1345 s) and the runtime of the Standard Blocker is 1 h 26 min 48 s (5208 s) on the large input set of 10^7 tuples.

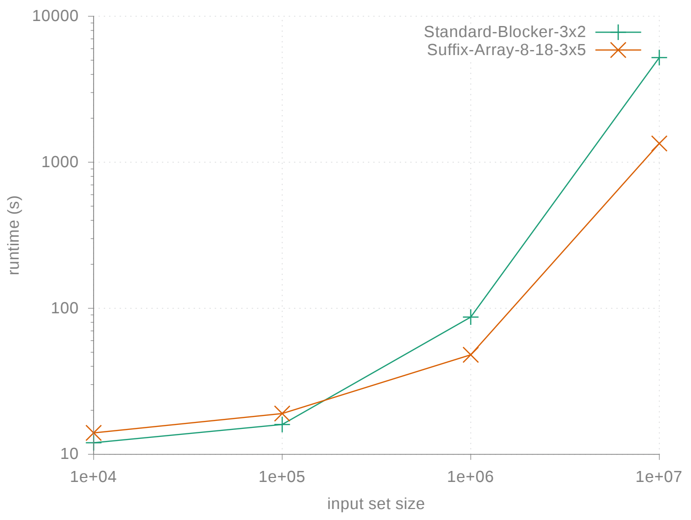


Figure 5.9.: Comparison of two different Blockers on the large files of 10^7 tuples. The cluster configuration is 4-worker. The dirtiness of the input data is 0.1.

5.3.5. Without Analyzing the Results

Analyzing intermediate results can be very time consuming. To get an impression on how the analytics part influences the runtime we compare the same pipeline with and without analytics. Figure 5.11 shows the standard pipeline with and without analytics on the configurations 1-worker and 4-worker. Due to technical reasons the gold standard is read in this experiment although it is not used. The results show that the analytics part of the pipeline accounts for about one half to one third of the overall runtime.

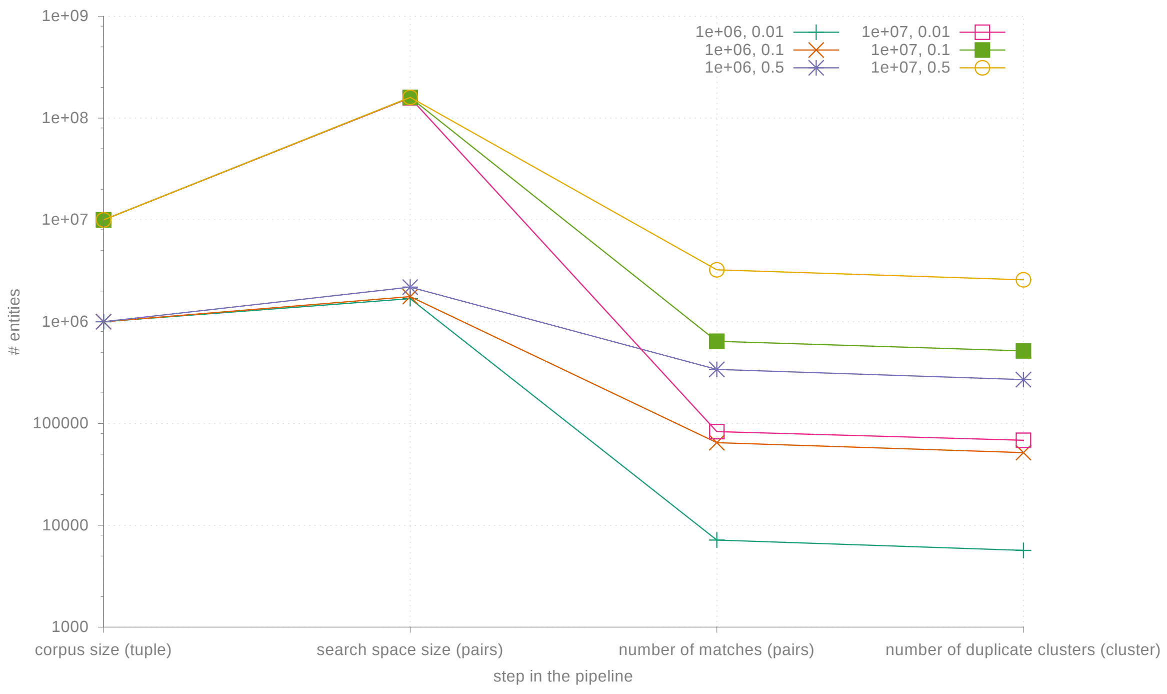


Figure 5.10.: Comparison of the intermediate result sizes of the Standard Blocker on the two different file sizes of 10^6 and 10^7. The dirtiness of the input data is 0.1.

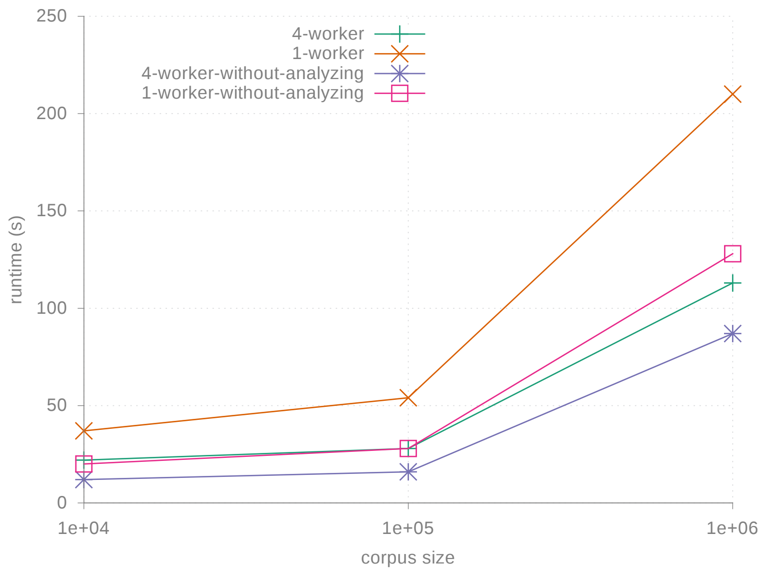


Figure 5.11.: Comparing the standard pipeline runtimes with and without analytics on the smallest cluster with one worker node and the largest with four worker nodes. The dirtiness of the data is 0.1.

5.3.6. On a Single Node

A Spark application can be launched in local mode without a cluster. This mode is mostly used for development and testing purposes, but it can also be used to transform a cluster application into a desktop application. To analyze how the process scales on a single machine, the duplicate detection process is run on the master node in local mode. HDFS was not used. Instead, the local file system was accessed directly. The driver memory is increased to 6 GB to equal the memory of the driver in local mode and a worker node in the small cluster. Figure 5.12 shows the runtimes for different file sizes. The file size of 10^7 is omitted because it is too large to be processed by a single node. On small amounts of data the machine in local mode is nearly as fast as the large cluster with four worker nodes. This is because of the lack of distribution overhead and in local mode the driver, master and all executors are executed in the same JVM. This is also the reason why the local mode is slower than the 1-worker configuration on larger input sets. Since in the 1-worker configuration driver and master are executed on the master node, the whole memory of 6 GB on the worker node can be used by the executors.

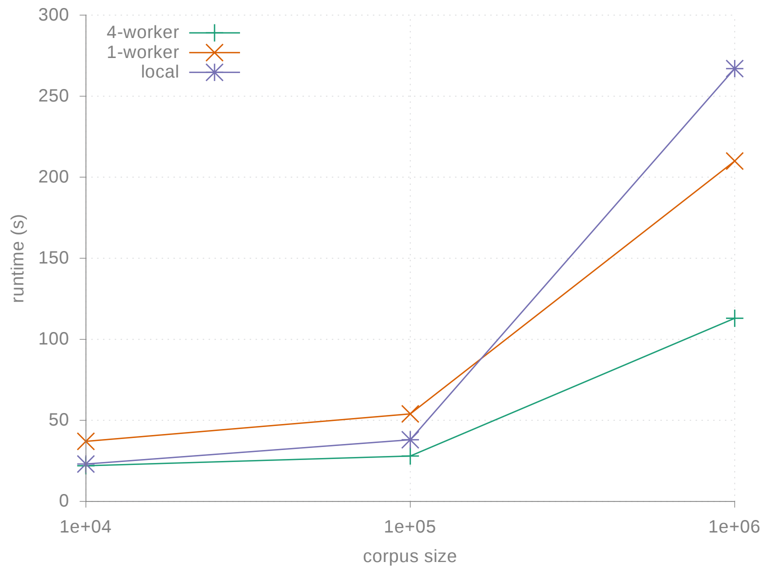


Figure 5.12.: Comparison of the standard pipeline running on a single node in local mode, the smallest cluster with one worker node and the largest with four worker nodes. The dirtiness of the data is 0.1.

5.3.7. Comparison with Dedoop

The comparison with Dedoop is quite difficult and vague. Like detailed out by Weis et al. [WNB06, p. 67] it is a challenge to benchmark duplicate detection approaches. This section highlights the problems of a real comparison. It is only done on basis of the

publication by Lars Kolb [Kol14, p. 85].
At first, we want to distinguish the differences of both setups. The size and shape of input data are different. Unfortunately, we could not obtain at least one of the two input sets of the Dedoop evaluation. The hardware differs. Dedoop is evaluated with a cluster of up to 100 virtual machine instances c1.medium of the EC2. A c1.medium instance has two cores and 1.83 GB memory. A machine of our cluster has four cores and 8 GB memory and no virtualization is used. The network transfer rate of the c1.medium instances is not listed on the amazon website. Dedoop uses MapReduce and its algorithms are highly optimized to the map reduce paradigm by doing load balancing. We do not do any explicit load balancing. These are the main differences of both setups.
It takes Dedoop about 110 minutes to run a Standard Blocker on the input set DS2 of $1.4 \cdot 10^6$ tuples with 20 nodes [Kol14, p. 85 figure 4.13.b]. The search space has a size of $6.7 \cdot 10^9$. On another input set DS1 of a size of $1.1 \cdot 10^5$ tuples and a search space size of $3 \cdot 10^8$ it takes Dedoop about 7 minutes to do the blocking. The runtimes are assumed to be blocking only, since there is no information whether other steps are included in the runtime. SddF requires 52 seconds to run Standard Blocking on the input set size of 10^7 tuples with a dirtiness of 0.1 on a cluster consisting of 4 worker nodes. The search space size is $1.6 \cdot 10^8$. The runtime of SddF does not include writing the result to disk. The results are only indications that SddF is faster than Dedoop. For a detailed comparison the setups have to be aligned.

5.3.8. Comparison with PPJoin+ MapReduce Implementation

This comparison is also very rough and difficult. It refers to the publication “Efficient Parallel Set-Similarity Joins Using MapReduce” [VCL10]. The runtime of the implementation compared to Dedoop is very low. For a self join of an input set size of $1.2 \cdot 10^7$, the runtime is about 300 s, which is fast compared to Dedoop. But there are several aspects why a comparison is difficult.
The hardware is highly optimized for the specific setup. There are four hard disks in each node. That is one per core and executor, which is the optimum. It is not clear which kind of hard disk was used. Since Hadoop make massive usage of the hard disk it would be essential to know whether Solid State Disks (SSDs) or magnetic hard disks have been used. Each node has a dual Gbit/s Network Interface Card (NIC). It is not clear whether they used both. Each node has 12 GB of RAM. They used a cluster of maximum 10 nodes. The description of the input data set lacks important information. The number of duplicates in the input sets and the size of the search spaces are both not described. Also quality measures like precision and recall, which indicate a representative experiment, are not mentioned. Moreover, there is no clustering step which resolves contradictions.
Our cluster consisting of 4 worker nodes needs 22 min 25 s (1345 s) to apply the whole

duplicate detection pipeline on an input data set of 10^7 tuples with a dirtiness of 0.1, utilizing a Suffix Array Blocker. For the results see Figure 5.9. It is very likely that the runtime of SddF can be further optimized. Take into consideration that PPJoin+ is a optimized implementation of a single class of algorithms and not a generic framework like SddF is. The next sections provides some thoughts about optimizations.

5.3.9. Optimizations Left Out

The framework SddF is a generic framework and therefore not optimized for a single use case. The following aspects could have been done to reduce the runtime. Some of them are addressed in Chapter 6.

1. Remove feature language:
   The feature language is part of the tuples, but is not part of the similarity calculation. It could have been omitted in the parsing step.

2. Read input set and gold standard in one single step:
   To provide a flexible API, the parsing of the input set and gold standard are done in two different steps, since they are often provided in different files and formats. In our case both input set and gold standard are in the same file and could have been parsed in the same step.

3. Replace tuples by ids when possible:
   During the processing of the whole pipeline there are always tuple instances in memory. After the similarity calculation they could have been replaced by an id placeholder. That would have reduced the memory consumption. A drawback would be the loss of accessing tuple features for debug or printing purposes especially in the interactive mode. This is also valid for the gold standard, which is also a set of tuple pairs and not only id pairs.

4. Remove similarity vector after classification:
   At the moment the similarity vector is not used during the clustering since there is only a transitive closure clustering available. For other algorithms like center clustering the vector is passed to the clustering. Even center clustering would not need the entire vector, but an aggregated value like the mean or median of all dimensions.

5. Use Kryo serialization:
   When data is transfered via network it gets serialized. The default serializer Spark uses is the standard Java serializer. The performance of the Java serializer is not that efficient compared to other serializers. The Kryo serializer is in general much faster and serializes in a more compact format.

6. Optimized usage of Spark cache:
   The Spark cache is a fraction of the memory where RDDs are stored which have been explicitly cached. The fraction can be configured to optimize the performance. RDDs can be removed from the cache explicitly, when they are not needed anymore. For instance, the search space could be removed from cache, if the similarity calculation have taken place and the result is cached.

7. Load balancing:
   Other approaches like Dedoop [Kol14] do load balancing to equally distribute the load among the cluster nodes. An explicit load balancing is not implemented yet, but the intermediate result type is always a tuple pair and not a set of tuples. This is an overhead, since all pairs of a set consume more space than a single set. Nevertheless, the pair guarantees a fixed size, which is better for load balancing purposes.

6. Conclusion

6.1. Summary

The thesis objective was to theoretically sum up distributed duplicate detection and on this basis to design, implement and evaluate the scalable duplicate detection framework SddF. Scalable explicitly means to scale up and down. With Apache Spark a shared nothing in memory distributed computing framework was chosen, which suits the scalability requirement. The second requirement was to create a modular framework, which can be easily extended by new algorithms. This was achieved by a pipelining architecture which results in a small linear typesafe pipelining framework called PipE. To evaluate the framework a new base data set was extracted from the Musicbrainz database, which was used to derive several data sets of different sizes and shapes from.
The evaluation focus lied on space and time. The different parameters varied are the input set size (10^4 – 10^7), the dirtiness of the data (0.01 – 0.5), the number of worker nodes (1 – 4) and the indexing algorithm. The evaluation showed that the framework scales with respect to the number of worker nodes on input sets of up to 10^6 tuples. Another result is, SddF bridges the gap between desktop computers and clusters. SddF is executable on a single desktop computer and on a large cluster without changing the code base. Due to Sparks in-memory data structure, SddF also runs performant on a single desktop machine. The evaluation also showed that it is possible to process large input set sizes of up to 10^7 tuples with a dirtiness of the data of 0.1 on a small cluster of only 4 worker nodes in 22.5 minutes. To analyze the scalability on such large input sets our cluster, consisting of 4 worker nodes, was not large enough. Another benefit is the exploratory application of SddF. This exploratory mode makes use of the Spark shell, which enables the user to connect the shell to a running cluster. Therefore, it is possible to interactively start duplicate detection pipelines and analyze intermediate results or compare different algorithms on large amounts of data.
Moreover, Scala and Spark offer many improvements compared to Java and Hadoop MapReduce. Due to the Scala and Spark API it is possible to write concise code. The Spark RDD API is similar to the one a non distributed collection would offer and it is no problem to test and execute a Spark or SddF application on a local computer.
Since the implementation was focused on a generic modular implementation and a convenient API, the optimization of the runtime was not always in the focus. Many possible

optimizations have been suggested in Section 5.3.9. In general, the new framework and Spark as a platform is promising and could be a new basis for parallel duplicate detection. SddF could also be used as a benchmarking platform for duplicate detection algorithms, since benchmarking is difficult in the duplicate detection domain. To achieve this, additional algorithms have to be implemented. Since it is an open source project, everyone is invited to contribute.

6.2. Criticism

It would have been beneficial to evaluate the framework on a much larger cluster than 4 worker nodes, but there was not the possibility to prepare a larger cluster. Especially to analyze the runtimes of the large input set size of 10^7 a larger cluster is required. A detailed comparison with other related projects would have been great, but comparisons of distributed duplicate detection frameworks are very difficult since there are so many degrees of freedom. Also a comparison with a sequential duplicate detection application like Febrl would have been interesting. It is difficult to align the setups of the different systems to provide a basis on which a fair comparison can take place. Additionally, it would have been beneficial to monitor the cluster during the experiments to analyze the load, memory consumption, disk and network usage.
A promising alternative would have been to implement SddF in Python to make use of Febrl[21] and scikit-learn[22]. This option has been left out because the best supported programming language of Spark is Scala, the Spark shell only supports Scala and the author prefers typesafe languages.

6.3. Future Prospects

There are four main topics for the future of this project. These are optimization, extending algorithms, usability and automation. The main point to optimize the memory consumption would be the implementation of a lookup facility for tuples. That is the requirement to optimize the intermediate result sizes without losing the possibility to access and output single tuple features in an arbitrary pipeline step. After doing this, it would be possible to only store tuple IDs instead of tuple instances. Instead of tuple pairs the gold standard could only contain tuple IDs. The same holds for the tuple pairs after the calculation of the similarity vectors. A promising candidate to implement such a distributed lookup facility is Hazelcast[23], a distributed lightweight key value store implemented in Java. Also

[21]`http://sourceforge.net/projects/febrl/`
[22]Python machine learning library, `http://scikit-learn.org`
[23]`https://hazelcast.org`

the possibility to unpersist RDDs in a pipeline would be a good feature to keep the Spark cache clean.
Since there is only one naive clustering algorithm, more algorithms like center clustering should be adapted to the Spark API. It would also be interesting to implement a clustering algorithm based on page rank to find cluster centers and afterwards applying a center clustering. The explicit handling of missing features should be considered since MLlib does not support them at present.
Very helpful would be the implementation of a graphical user interface, preferably a web interface with a live plotting facility maybe using “Wisp”[24]. To improve the practical benefit it would be useful to implement manual review and labeling pipelines which can be used to manually generate training data without the need for a gold standard.
Another target would be to implement an automated optimization like grid search with algorithm and parameter sets. This would decrease the manual work to configure and execute different pipelines. Long term plan would be to automate the creation of the deduplication pipeline, like the “MLbase” project [KTD+13] of the AMPlab Berkeley is going to do for arbitrary machine learning pipelines. That means the automation of analyzing the input data and choosing algorithms and parameters to solve the duplicate detection problem.

[24]`https://github.com/quantifind/wisp`

Bibliography

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*, pages 734–746, 2013.

[ACG02] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating Fuzzy Duplicates in Data Warehouses. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 586–597. Morgan Kaufmann, 2002.

[AO05] Akiko N. Aizawa and Keizo Oyama. A Fast Linkage Detection Scheme for Multi-Source Information Integration. In *2005 International Workshop on Challenges in Web Information Retrieval and Integration (WIRI 2005), 8-9 April 2005, Tokyo, Japan*, pages 30–39. IEEE Computer Society, 2005.

[BGG+06] Omar Benjelloun, Hector Garcia-Molina, Heng Gong, Hideki Kawai, Tait Eliott Larson, David Menestrina, and Sutthipong Thavisomboon. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 37. IEEE Computer Society, 2006.

[BGH11] Guilherme Dal Bianco, Renata de Matos Galante, and Carlos A. Heuser. A fast approach for parallel deduplication on multicore processors. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1027–1032. ACM, 2011.

[BMC+03] Mikhail Bilenko, Raymond J. Mooney, William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. Adaptive Name Matching in Information Integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* Wiley, 1996.

[CCH04] Peter Christen, Tim Churches, and Markus Hegland. Febrl – A Parallel Open Source Data Linkage System. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Advances in Knowledge Discovery and Data Mining*, volume 3056 of *Lecture Notes in Computer Science*, pages 638–647. Springer Berlin Heidelberg, 2004.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 313–324. ACM, 2003.

[Chr12] Peter Christen. *Data matching.* Springer, 2012.

[CRP$^+$10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. *SIGPLAN Not.*, 45(6):363–375, June 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation.* USENIX Association, 2004.

[FS69] I. P. Fellegi and A. B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.

[Gol80] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs.* Computer science and applied mathematics. Academic Press, New York, 1980.

[HCB98] Lawrence Hall, Nitesh Chawla, and Kevin W. Bowyer. Combining Decision Trees Learned in Parallel. In *In Working Notes of the KDD-97 Workshop on Distributed Data Mining*, pages 10–15, 1998.

[Hil15] Kai Hildebrandt. Obtaining large scale test data from real world datasets for duplicate detection using Apache Spark. Master's thesis, University of Hamburg, 2015. not available yet.

[HK06] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HM09] Oktie Hassanzadeh and Renée J. Miller. Creating Probabilistic Databases from Duplicated Data. *The VLDB Journal*, 18(5):1141–1166, October 2009.

[Hol15] Alex Holmes. *Hadoop in Practice.* Manning Publications Co., Greenwich, CT, USA, second edition, 2015.

[HS95] Mauricio A. Hernández and Salvatore J. Stolfo. The Merge/Purge Problem for Large Databases. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 127–138. ACM Press, 1995.

[HS98] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

[KKH$^+$10] Toralf Kirsten, Lars Kolb, Michael Hartung, Anika Groß, Hanna Köpcke, and Erhard Rahm. Data Partitioning for Parallel Entity Matching. *CoRR*, abs/1006.5309, 2010.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly, 01 2015.

[KL07] Hung-sik Kim and Dongwon Lee. Parallel linkage. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 283–292. ACM, 2007.

[KL10] Hung-sik Kim and Dongwon Lee. HARRA: Fast Iterative Hashed Record Linkage for Large-scale Data Collections. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 525–536, New York, NY, USA, 2010. ACM.

[Kol14] Lars Kolb. *Effiziente MapReduce-Parallelisierung von Entity Resolution-Workflows*. PhD thesis, University of Leipzig, 2014.

[KTD$^+$13] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[NH10] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. Morgan and Claypool, 2010.

[PMM+02] Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart J. Russell, and Ilya Shpitser. Identity Uncertainty and Citation Matching. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, pages 1401–1408. MIT Press, 2002.

[SB02] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 269–278. ACM, 2002.

[SG14] Sherif Sakr and Mohamed Medhat Gaber. *Large Scale and Big Data - Processing and Management.* CRC Press, 2014.

[TLX13] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal mapreduce algorithms. In *Proceedings of the 2013 international conference on Management of data*, pages 529–540. ACM, 2013.

[Tuk93] John W. Tukey. Exploratory Data Analysis: Past, Present, and Future. 1993.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In 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*, pages 495–506. ACM, 2010.

[VME03] Vassilios S. Verykios, George V. Moustakides, and Mohamed G. Elfeky. A Bayesian decision model for cost optimal record matching. *VLDB J.*, 12(1):28–40, 2003.

[WNB06] Melanie Weis, Felix Naumann, and Franziska Brosy. A duplicate detection benchmark for XML (and relational) data. In *SIGMOD 2006 Workshop on Information Quality for Information Systems (IQIS), Chicago, IL, 2006.*, 2006.

[XWLY08] Chuan Xiao, Wei Wang, 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, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140. ACM, 2008.

[ZCD$^+$12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZGGW11] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *IPDPS Workshops*, pages 1112–1121. IEEE, 2011.

Acronyms

API	Application Interface
ASCII	American Standard Code for Information Interchange
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
EC2	Amazon Elastic Compute Cloud
EDA	Exploratory Data Analysis
GCD	Google Cloud Dataflow
HA	High Availability
HDFS	Hadoop Distributed File System
HiveQL	Hive Query Language
HQL	Hive Query Language
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MPI	Message Passing Interface
NIC	Network Interface Card
OCR	Optical Character Recognition
PDF	Portable Document Format

RDD	Resilient Distributed Dataset
RMI	Remote Method Invocation
SDK	Software Development Kit
SSD	Solid State Disk
SVG	Scalable Vector Graphics
SVM	Support Vector Machine
TU	Technical University
VM	Virtual Machine
YARN	Yet Another Resource Negotiator

List of Figures

List of Tables

List of Listings

A. Appendix

A.1. Musicbrainz

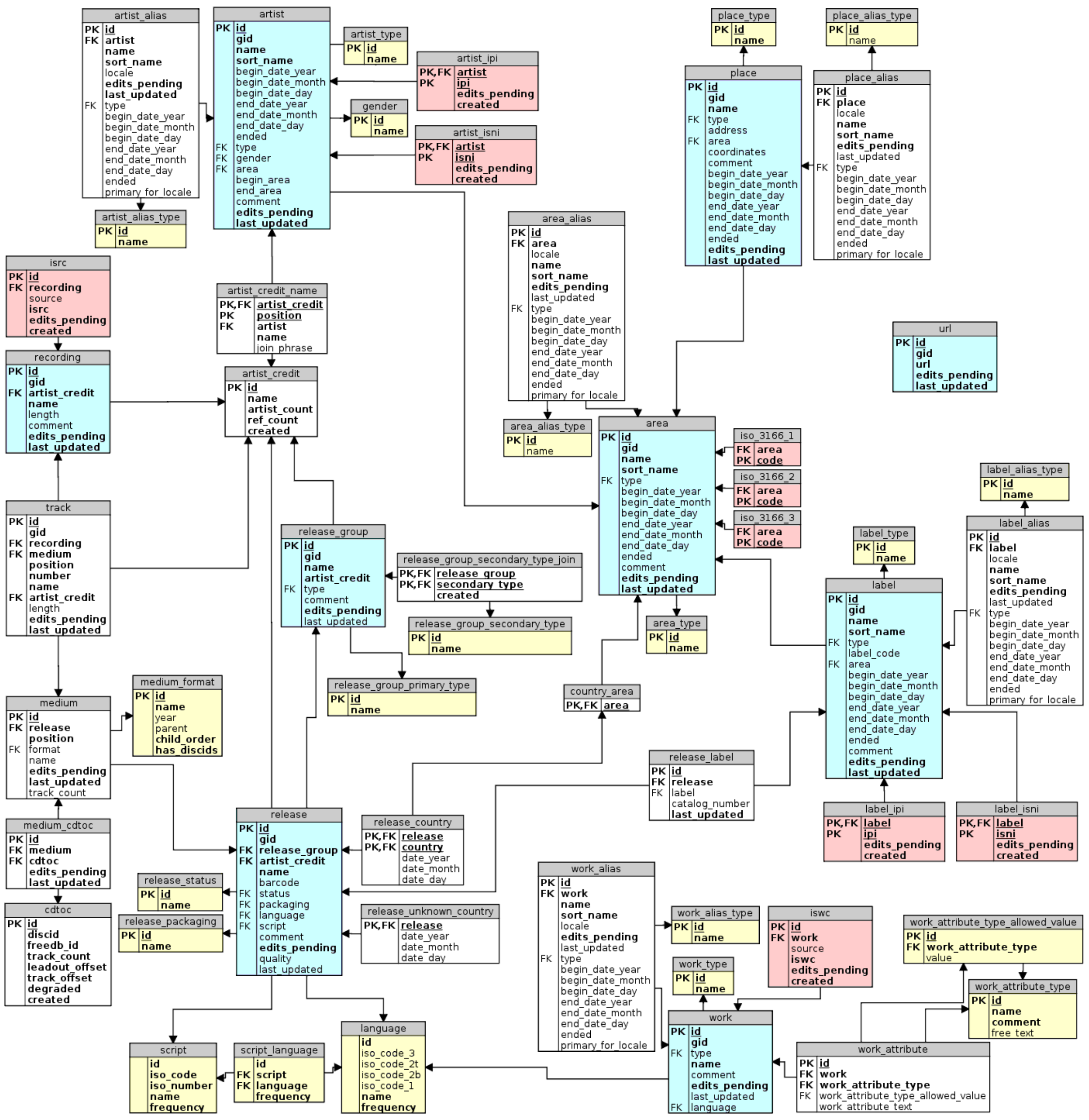

artist_alias: PK id, FK artist, name, sort_name, locale, edits_pending, last_updated, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, primary_for_locale
artist: PK id, gid, name, sort_name, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, FK type, FK gender, FK area, begin_area, end_area, comment, edits_pending, last_updated
artist_type: PK id, name
artist_ipi: PK,FK artist, PK ipi, edits_pending, created
gender: PK id, name
artist_isni: PK,FK artist, PK isni, edits_pending, created
artist_alias_type: PK id, name
isrc: PK id, FK recording, source, isrc, edits_pending, created
artist_credit_name: PK,FK artist_credit, PK position, FK artist, name, join_phrase
recording: PK id, gid, FK artist_credit, name, length, comment, edits_pending, last_updated
artist_credit: PK id, name, artist_count, ref_count, created
area_alias: PK id, FK area, locale, name, sort_name, edits_pending, last_updated, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, primary_for_locale
area_alias_type: PK id, name
place_type: PK id, name
place_alias_type: PK id, name
place: PK id, gid, name, FK type, address, FK area, coordinates, comment, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, edits_pending, last_updated
place_alias: PK id, FK place, locale, name, sort_name, edits_pending, last_updated, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, primary_for_locale
url: PK id, gid, url, edits_pending, last_updated
area: PK id, gid, name, sort_name, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, comment, edits_pending, last_updated
iso_3166_1: FK area, PK code
iso_3166_2: FK area, PK code
iso_3166_3: FK area, PK code
track: PK id, gid, FK recording, FK medium, position, number, name, FK artist_credit, length, edits_pending, last_updated
release_group: PK id, gid, name, artist_credit, FK type, comment, edits_pending, last_updated
release_group_secondary_type_join: PK,FK release_group, PK,FK secondary_type, created
release_group_secondary_type: PK id, name
release_group_primary_type: PK id, name
area_type: PK id, name
country_area: PK,FK area
label_alias_type: PK id, name
label_type: PK id, name
label_alias: PK id, FK label, locale, name, sort_name, edits_pending, last_updated, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, primary_for_locale
label: PK id, gid, name, sort_name, FK type, label_code, FK area, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, comment, edits_pending, last_updated
medium_format: PK id, name, year, parent, child_order, has_discids
medium: PK id, FK release, position, FK format, name, edits_pending, last_updated, track_count
release_label: PK id, FK release, FK label, catalog_number, last_updated
label_ipi: PK,FK label, PK ipi, edits_pending, created
label_isni: PK,FK label, PK isni, edits_pending, created
medium_cdtoc: PK id, FK medium, FK cdtoc, edits_pending, last_updated
release: PK id, gid, FK release_group, FK artist_credit, name, barcode, FK status, FK packaging, FK language, FK script, comment, edits_pending, quality, last_updated
release_country: PK,FK release, PK,FK country, date_year, date_month, date_day
release_status: PK id, name
release_packaging: PK id, name
release_unknown_country: PK,FK release, date_year, date_month, date_day
cdtoc: PK id, discid, freedb_id, track_count, leadout_offset, track_offset, degraded, created
work_alias: PK id, FK work, name, sort_name, locale, edits_pending, last_updated, FK type, begin_date_year, begin_date_month, begin_date_day, end_date_year, end_date_month, end_date_day, ended, primary_for_locale
work_alias_type: PK id, name
iswc: PK id, FK work, source, iswc, edits_pending, created
work_attribute_type_allowed_value: PK id, FK work_attribute_type, value
work_type: PK id, name
work_attribute_type: PK id, name, comment, free_text
work: PK id, gid, FK type, name, comment, edits_pending, last_updated, FK language
work_attribute: PK id, FK work, FK work_attribute_type, FK work_attribute_type_allowed_value, work_attribute_text
script: id, iso_code, iso_number, name, frequency
script_language: id, FK script, FK language, frequency
language: id, iso_code_3, iso_code_2t, iso_code_2b, iso_code_1, name, frequency

Figure A.1.: Schema of the Musicbrainz database.

A.2. Other Related Projects

There are multiple other related projects. Especially the Hadoop ecosystem is full of projects addressing similar problems. The following list is just an excerpt and not intended to be complete.

Weka is a mature project of the University of Waikato in New Zealand which started in 1992. With Weka 3 the project switched to Java and an open source license. Core features are visualization and algorithms for data analysis and a graphical user interface. Weka was originally designed as a single computer application, but there are attempts to enable Weka to process huge amounts of data. Since October 2013 there is a package available called "distributedWekaBase" which provides basic functionality for a map reduce like backend. The Weka developer also provided a first platform specific implementation atop of Hadoop called "distributedWekaHadoop".

Apache Storm became an Apache top level project in September 2014. Apache Storm is a distributed real time data processing framework. Main feature is the strong guarantee on the processing of data.

Apache Flink , formerly known as Stratosphere, is an Apache Incubator project since April 2014 which was also the date of the renaming. Stratosphere started in 2009 at the Technical University (TU) Berlin. Flinks last release version is 0.8.1. Superficially seen Apache Flink is similar to Apache Spark, but Flink is quite different and introduces some new concepts. Flink tries to bring in-memory data processing and database optimization techniques together. It also introduces a new high level API called delta-iterations to express iterative algorithms, which is far better suited for optimization purposes.

Apache Tez is a Apache top level project since Juli 2014. It is a framework to build large scale data processing applications on top of Apache YARN. Main feature is the processing of a complex DAG of tasks.

Apache Zeppelin became an Apache incubator project in December 2014. It is a web notebook for interactive data analytics. Zeppelin combines an interactive shell like input terminal with data visualization.

Apache Pig is a high level abstraction layer on top of Hadoop MapReduce. Applications are written in the procedural language Pig Latin. Pig tries to abstract from the low level map reduce paradigm with high level functionality.

Apache Hive is a data warehouse API build on top of Hadoop MapReduce. The declarative language Hive Query Language (HiveQL), similar to SQL, is used to formulate queries.

Cascading is an abstraction layer atop on Hadoop MapReduce to hide MapReduce specific complexity. It provides a high level API to make programming of map reduce jobs more intuitive and efficient.

Scalding is a Scala wrapper for Cascading. It is developed by Twitter.

Summingbird is an open source project developed by Twitter. Summingbirg incorporates Scalding and Storm to build a stream and batch processing framework. The API is accessible via Scala and Java.

Apache Crunch is a top level Apache project since February 2013. It is an abstraction layer atop on Hadoop MapReduce like Pig and Hive are. Crunch does not focus on data analysts with restricted programming skills. Instead, it targets real developers. It aims to be a high performance framework which let the developer decide when to use low level map reduce operations and when to use high level operations. There are Scala bindings available for Crunch.

Apache Oozie Apache Oozie is a DAG scheduler for Hadoop MapReduce jobs. It is implemented as a Java web application. All processing is done by the underlying Hadoop MapReduce. Oozie keeps track of the processing state via callback URLs or via polling.

A.3. Hadoop

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
```

```
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Listing A.1: Word count example with Apache Hadoop MapReduce.[26]

A.4. Content of the DVD

- Master’s Thesis
- Apache Spark 1.3.1
- SddF
- Experiments
- Results
- Cluster management scripts

[26]Source: http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v2.0

- Cluster node configuration for master and slaves
- Setup instructions for the cluster nodes and IDE
- Musicbrainz data sets
- Dapo extension to generate the input data sets

A.5. Used Software

Many thanks to all developers of free and open source software. Besides Yed this thesis was created by only using open source software. Special thanks goes to the whole Linux community, which is doing such a great work. The main programs that where used to create this thesis are listed below.

Gnu/Linux Operating system (Debian, Ubuntu, openSUSE)

Git Decentralised Version Control System

Scala Scalable programming language for the JVM

Apache Spark shared nothing distributed in-memory computing framework

Apache Spark MLlib Machine learning library for Spark

Apache Spark GraphX Graph processing library for Spark

Python dynamic typed scripting language

CSSH cluster ssh client

Latex Template of Stefan Macke Latex template for master thesis

Kile Latex editor

KBibtex Bibtex reference management software

Okular Portable Document Format (PDF) viewer

Libre Office Draw Scalable Vector Graphics (SVG) drawing program

Gnuplot feature rich SVG plotting software

yEd SVG graph editor

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 7. September 2015

NIKLAS WILCKE

Erklärung zur Veröffentlichung in der Bibliothek

Ich bin ausdrücklich damit einverstanden, dass die Arbeit in analoger und digitaler Form in den Bestand der Bibliothek aufgenommen wird. Zudem stimme ich einer online Publizierung zu.

Hamburg, den 7. September 2015

Niklas Wilcke