Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Efficient Record Linkage with MapReduce in Very Large Data Sets

A dissertation submitted to the Karlsruhe University of Applied Sciences in accordance with the requirements for award of the degree of *Bachelor of Science (B.Sc.)* in the Department of Computer Science.

Stefan Sperber

stefan.sperber@gmail.com

September 2011

Abstract

Linking entries that represent the same entity in data sets is a common task when working with data. Applications include merging two or more data sets that do not share a key relationship or finding duplicate entries in a local data set. Websites might rely on data from outside sources to extend their user experience, for example, with additional information or links to an affiliate's service.

When data sets are linked, a few things have to be considered. The entries in both sets might not share the same format, the number and meaning of fields might differ, and only for a certain number of entries a corresponding partner entry might exist. Also, modern databases store significant amounts of data. Although the question of how to find possible matches has extensively been addressed in past research, it still remains valid due to the ever-growing scale of the inputs. It is not always clear how proposed methods handle real-life data and how they perform with large data sets.

This dissertation takes a close look at common practices for automatically finding matches in two data sets of different origin. To achieve this, both established and recently published methods will be evaluated with small but well-crafted data sets. Hereby, a combination of methods which produces the best results is to be found. This combination is then applied to MapReduce, a programming model for distributed processing of large data sets, in order to design a prototypical implementation.

Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the Karlsruhe University of Applied Sciences. The work is original, except where indicated by special reference in the text, and no part of the dissertation has been submitted for any other academic award. Any views expressed in the dissertation are those of the author.

Karlsruhe, on the 30th September 2011

Contents

1 Introduction

Every day we leave hundreds of data footprints, be it online or in real life. For example, social networking websites encourage users to share impressions from their private life with others. Thus, when browsing the Internet we are surrounded by e-mails, photos, or status updates that we, or others have created. The websites we frequent register visits in log files. Data retention laws may make it mandatory for Internet service providers to store certain Internet traffic over a period of time. In other parts of life, public transport authorities may collect data from commuters with an electronic ticket to identify possible future developments for transport networks. Super markets often reward customers for using loyalty cards, as these make individual shopping preferences visible.

Today, however, one does not have to remain a passive producer of data but can easily become an active consumer. Over recent years, large-scale data analysis has been "democratised" by an abundance of publicly available data sets, cheap infrastructure for storage and processing, and tools for analysis. Open data movements have led government agencies to publish their data for everyone (data.gov[1] and data.gov.uk[2] are just two examples), while websites like Infochimps[3] aggregate links to data sets from many different fields. All this data holds amazing possibilities that just wait for data scientists or interested individuals to be discovered.

A common task when several data sets are connected for analysis is to identify similar entries. This is known as *record linkage*, a challenging task when no direct relation (e.g. a key relationship) between data sets exists. Only after a thorough analysis, the full information content of the examined data can be uncovered. The newly found insights then can form the foundation to inspire and interest others. An appealing visualisation[4], for example, will help to make sense of complex data flows much better than just dry numbers.

[1]`http://www.data.gov`
[2]`http://www.data.gov.uk`
[3]`http://www.infochimps.org`
[4]`http://www.infosthetics.com/`

1.1 Common Techniques for Different Applications

Traditionally, record linkage is used for merging databases [19] and for quality control of census data [34], but many different applications can benefit from advancements made in record linkage and vice versa, for example content recommendation [20] or molecular biology [26]. In the following we present several use cases.

Information retrieval: By correlating traffic statistics and crime rates with apartment listings when looking for accommodation, safe neighbourhoods can be identified that are close to the workplace or an urban centre.

Data integration: After a company acquisition, new customer data needs to be integrated into an existing customer relationship management database. Difficulties can arise from the data structure. For example, one database might store customer names in one field ("Joe Bloggs"), while the other database stores surnames and first names separately ("Bloggs", "Joe"). A simple solution to this is to concatenate first and last names in the right order prior to inserting a new entry. Next, before the integrated data can be used to contact customers, duplicate entries have to be identified. These may result from the integration process or may have already been present in the original databases.

Specialised tools for data integration exist, for example Talend Open Studio[5]. It has to be evaluated, however, how a secondary tool can be used in existing, automated workflows if data integration is a reoccurring problem.

Search-term refinement: Many search engines provide a “Did you mean” feature, which recommends other spellings for keywords. For this, the query needs to be parsed, similar terms have to be found, and a response has to be built and sent back to the user, all within a very short period of time.

1.2 Scaling the Problem

The ever-growing scale of data collections introduces another level of complexity. Today's data sets often consist of millions or billions of entries, and are usually too big to be kept in memory on one computer. For example, the *Million Song Dataset*[6] is a collection of audio features and metadata that were extracted from popular music tracks; it takes up about 280 gigabytes of space. With these dimensions, just downloading, importing, and storing data introduce a considerable amount of work, let alone later processing.

[5]`http://www.talend.com`

[6]`http://labrosa.ee.columbia.edu/millionsong/`

The term *Big Data* has been used to refer to this situation. It is not well-defined but rather a vague way of describing data that has become so large that classic storage and processing techniques reach their limit. In this context, MapReduce has received much attention as it offers one solution for data-centric applications to conquer Big Data. Computation is distributed over a cluster of machines where each node only works on a small aspect of the problem.

But even with new thinking models and tools to conquer increased scale, the well-known algorithms for data analysis and linking still will have to be adapted. Some algorithms might not be expressible in the MapReduce programming model, others might need to be modified before they can be employed. Performance might end up being different than before.

1.3 Dissertation Outline

The remainder of this dissertation is structured as follows. We define and discuss the problem that we are planning to solve in Chapter 2 in greater detail. Chapter 3 gives a general introduction to record linkage. We proceed with the realisation in Chapter 4 where we carefully select and implement techniques for each step of the record linkage process. For this, we can resort to the theoretical foundations we have laid before. We also create data sets for the following evaluation. Chapter 5 shows how each chosen technique is evaluated separately. This allows us to identify several combinations of techniques that perform best for our data. In Chapter 6 we introduce MapReduce and its open source implementation Hadoop, and study how the combinations we found in small-scale performs when adapted to a distributed environment. We conclude in chapter 7 by reviewing all our findings, and discuss future work.

2 Problem Statement and Analysis

For this dissertation, we have taken an interest in what existing techniques for record linkage can be used to match large data sets. We aim to identify approaches for MapReduce that can reliably and efficiently identify entries in two data sets that refer to the same real-world entity. Our research and work was supported by online music recommendation service Last.fm[1].

This chapter states and defines the problem that we are planning to solve. First, we briefly introduce Last.fm and then show in what parts of Last.fm data from outside sources regularly needs to be matched with local data. We conclude by presenting how we structure our work and what we focus our research on.

2.1 About Last.fm

Last.fm is a music recommendation service based in London and founded in 2004. Users automatically submit metadata for a song to Last.fm when they are listening to music on the computer or from a mobile device. Metadata includes, for example, artist name, album name and track title. The number of times a user has listened to a song or artist is registered, and can later be used to identify listening habits. By comparing listening habits to those of other users, similarities between artists or musical genres are found, and music recommendations based on individual taste can be given.

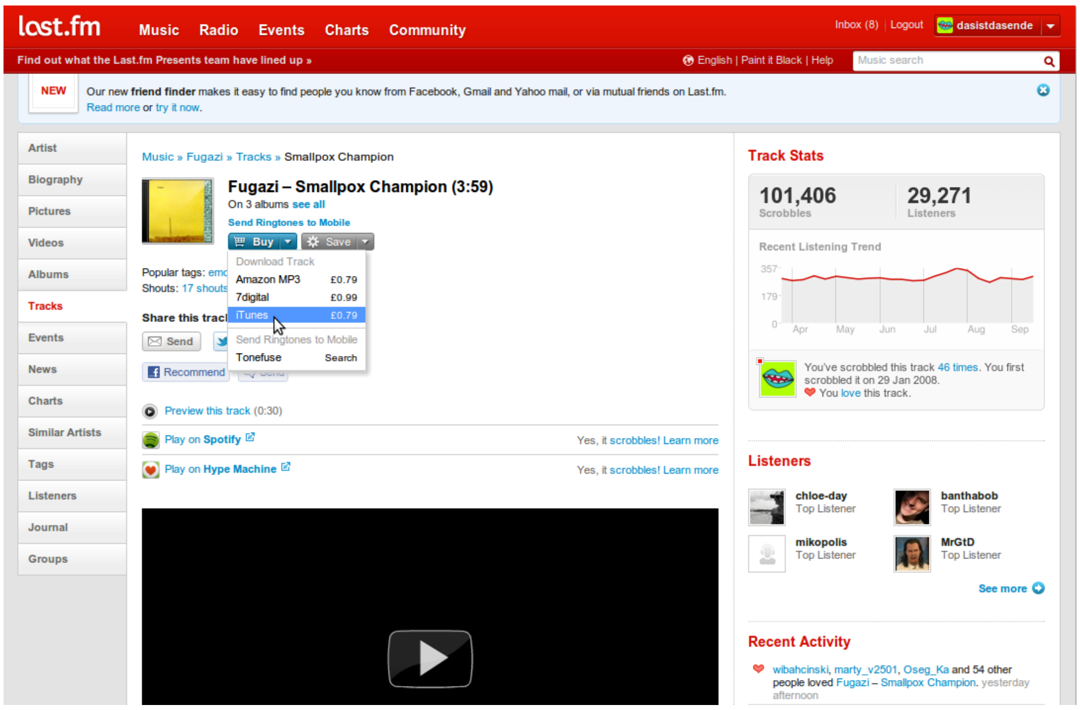
For our experiments, we used data sets from Last.fm's music catalogue and from their affiliates. Also, we could access the Hadoop cluster for testing MapReduce programs, and use additional infrastructure.

2.2 The Need for Record Linkage

We concentrated on one use case of record linkage for our research: identifying similar or equal entries in two data sets, as this is a common task for Last.fm.

Every track in Last.fm's music catalogue has a track page on the website. These pages present an overview with overall listener numbers and popularity, display comments left

[1]`http://last.fm`



NEW Our new friend finder makes it easy to find people you know from Facebook, Gmail and Yahoo mail, or via mutual friends on Last.fm. Read more or try it now.

Fugazi – Smallpox Champion (3:59)
On 3 albums see all
Send Ringtones to Mobile

Popular tags: emo
Shouts: 17 shouts
Share this track

Preview this track (0:30)
Play on Spotify — Yes, it scrobbles! Learn more
Play on Hype Machine — Yes, it scrobbles! Learn more

You've scrobbled this track 46 times. You first scrobbled it on 29 Jan 2008.
You love this track.

wibahcinski, marty_v2501, Oseg_Ka and 54 other people loved Fugazi – Smallpox Champion. yesterday afternoon

Figure 2.1: A track page on Last.fm. The page displays information on the track, and has links to external services by affiliates.

by the users, and recommend similar songs and artists. Also, additional links to affiliate services are presented. Figure 2.1 shows an example track page. Similar pages also exist for artists and albums.

Internally, affiliate links are differentiated in two types: *playlinks* and *buylinks*. Playlinks allow users listen to a song at an online streaming service, while buylinks point to online music stores where a song or album can be purchased in a digital format. Current affiliates include, among others, online music store Amazon MP3 and the iTunes Store, as well as Spotify and Hype Machine for streaming.

These links to external services are not retrieved from affiliates every time an album, track, or artist page is accessed, for example, by querying an API[2], as this would mean a lot of repeating requests would have to be made. Instead, affiliates provide their product data in a simple format for download, which is then matched to entries in Last.fm's music catalogue in a process called *affiliate ingestion*.

[2]Application programming interface

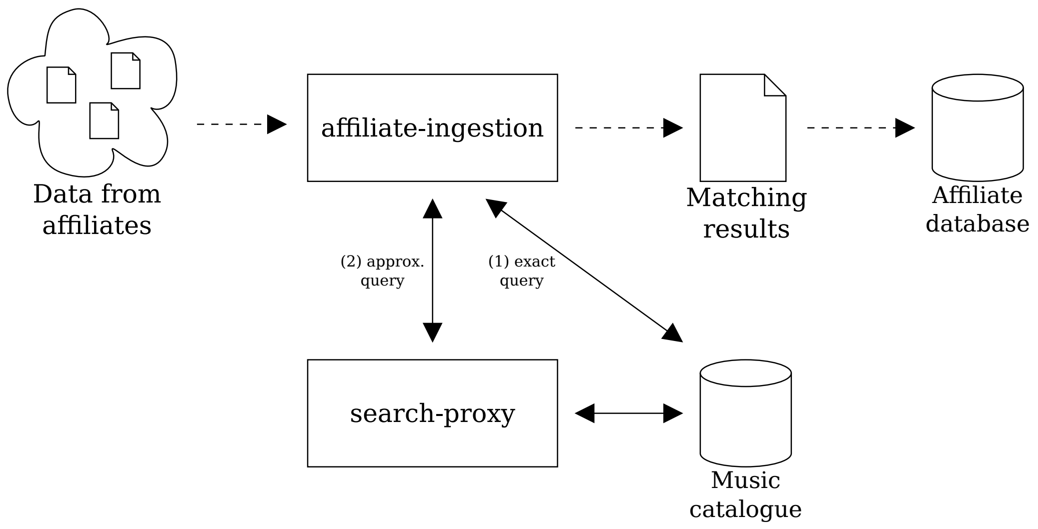


Figure 2.2: Simplified workflow for ingestion of data from affiliates at Last.fm.

2.2.1 Workflow of the Ingestion Process

The purpose of affiliate ingestion is to process product data from an outside party, match it to tracks and albums in the local music catalogue and then save the identified matches to a database. From there, they can be retrieved when links to affiliate services need to be displayed on the website later.

Despite involving the same steps, ingestion of buylinks and playlinks are separate projects with own codebases. MapReduce is used for ingesting buylinks, while playlink ingestion evolved from a multi-threaded prototype system. Common functionality already has been extracted to separate libraries, but there is still potential to further unify both projects. We therefore explain the ingestion process in broad terms that are valid for both projects, and leave out most details.

Figure 2.2 shows the workflow. First, a new set of data is downloaded. Its entries are parsed and brought to a comparable format. Artist names and track titles from the data set are used to query the music catalogue. If no exact match can be found, the ingestion system can revert to an internal service called *search proxy*, which supports approximate queries. It is used in many other parts of Last.fm, for instance, for a user-facing search function. For every matching entry pair, the track ID from the music catalogue and the link to the affiliate's service is saved to a intermediary file, which is imported to the affiliate database.

The current ingestion systems already achieve match rates of 80 per cent and above. Depending on the number of entries, matching takes between six and ten hours for one data set.

Affiliate	File size (MB)	Entry #
A	872	11,414,174
B	1,530	9,315,936
C	874	4,806,601
D	10,694	16,818,689
E	229	2,045,866

Table 2.1: Overview of properties of several data sets received from affiliates.

2.2.2 Scale and Structure of Data

The large scale of the music catalogue at Last.fm and the size of data received from affiliates introduces an increased level of complexity to finding matching entry pairs.

Music Catalogue

Over time, the music catalogue has steadily grown in size, and its schema went through multiple changes and extensions. As of writing, it holds roughly 520 million track titles, 48 million artist names, and 163 million album names.

Most of this data is user-submitted. When a user listens to a song that is not yet present in the music catalogue, a new entry is created. Its fields are populated with the received metadata. The metadata varies greatly in accuracy and completeness, and wrong entries or multiple entries for one song are common. These entries, however, are not deleted, as they may be submitted again in the future. Instead, a reference to a correct entry can be set for every track. Corrections are applied either automatically (e.g. for frequent misspellings), or manually by user vote.

Data from Affiliates

Affiliates provide their data in a simple text file: each line represents one database entry with values delimited by tab stops or other delimiters. Commonly, this format is referred to as *tab-separated values* or TSV. It is widely used for data exchange, as it is independent of products and vendors.

Every entry must at least have fields for artist name, track title and a link to the affiliate's service that can later be displayed. Other than that, every affiliate structures their data differently in terms of ordering of fields and extent. Table 2.1 lists properties of data sets that were received from affiliates and shows how numbers of entries as well as file sizes differ.

2.3 Aims and Contributions

Our aim is to learn how record linkage can be applied to large-scale data sets. By this, we will identify techniques that can be used to improve existing data-matching problems at online music recommendation service Last.fm.

We plan to make findings that are beneficial in the following ways:

1. A **higher match rate** can translate to more buying incentives on track, album, and artist pages and increases the number of users sent to services by affiliates.
2. A **more efficient workflow** will utilise fewer computing resources. Thereby, infrastructure becomes available again for other tasks sooner.

Although we are focused on techniques that suit data-processing demands at Last.fm, we are certain that the presented thoughts and results can be applied to a wide range of different scenarios with minor to no modification.

We will make the following contributions:

1. **Overview of the record linkage process** and all of its individual steps. We will study a number of different methods, for example, character- and token-based similarity metrics as well as vector-space techniques for finding groups of similar entries. These will be established methods but also come from current research.
2. **Identifying optimal combinations of techniques** that perform best for the data-matching tasks at Last.fm. For this, we will derive own data sets from real-world data, which will be used to evaluate all chosen techniques. The best-performing techniques from each step will be combined.
3. **Adaptation to MapReduce**. We will create a prototypical implementation of our findings that employs the MapReduce model. We will conduct final tests with large data sets. Afterwards, we will conclude by thoroughly reviewing our results.

3 Record Linkage

Record linkage is the task of identifying entries that represent the same logical entity in one or more data sets. Other names for the underlying problem include deduplication, entity resolution, data matching or duplicate detection.

When data is received from an outside source, its structure might not follow a common schema, even if all data is from the same domain. The number and meaning of fields may vary as well as the spelling and formatting used. Before the received data can be integrated, its structure needs to be changed in order to match the local schema. After merging, the data set might contain a number of duplicate entries. Two entries are considered duplicate if both describe the same real-world object — for instance, a person or a musical recording — but their representations are not identical. Duplicates affect and degrade the quality of a data set, as it becomes more difficult to retrieve a unified view when information is scattered over multiple entries.

Even without the need to link two data sets, a data set accumulates duplicates over time. Often, a new entry is created instead of extending or updating an existing one. Possible reasons for this are missing constraints that enforce distinct values for attribute fields, spelling errors when data is manually entered or poorly formatted fields, such as telephone numbers with missing country prefixes.

Table 3.1 illustrates this problem. The structure of the data is different, even though both tables are (artificial) digests from music catalogues. Some tracks are present in both tables, while each table also contains a number of unique tracks. The types and number of fields are different. We will use these tables as a basis to illustrate the effect of some of the techniques introduced over the course of this chapter.

We will start this chapter with a short, formal definition of the record linkage problem. Next, we introduce each of the steps involved in the process — normalisation, scoring and matching, and blocking — and their purpose.

3.1 Formal Explanation and Process Overview

Our main interest is in the application of record linkage, however, we first will give a brief formal overview of the process. For an extensive mathematical definition, see the work of

id	artistname	tracktitle	year
1	Arcade Fire	Une année sans lumière	2005
2	Jimi Hendrix	All Along The Watchtower	XXXX
3	Radiohead	Electioneering	'97
4	Bob Dylan	Like A Rolling Stone (2010 Mono Version)	1965
5	david bowie	life on mars?	1971
6	Jimi Hendrix	the wind cries mary (single)	1967

(a) Local music catalogue.

id	artist	track	album	length
1	Beatles	Day Tripper	NULL	2:48
2	Clash, The	Should I Stay Or Should I Go	Combat rock	3:15
3	RADIOHEAD	ELECTIONEERING	OK COMPUTER	3:49
4	Bob Dylan	Like A Rolling Stone	album	-1
5	David Bowie	Life On Mars (1999 Digital Remaster)	Best of Bowie (disc1)	3:44

(b) External music catalogue that needs to be integrated.

Table 3.1: Data sources used throughout this chapter.

Fellegi and Sunter [11] who first introduced the record linkage problem in mathematical terms in their article “A Theory for Record Linkage” in 1969.

Let R, S be two collections of entries, with $R = (r_0, \ldots, r_n)$ and $S = (s_0, \ldots, s_m)$. An entry consists of a number of attributes. In a database notion, the collections of entries are database tables, the entries are records stored in these tables and their attributes are fields. The record linkage task comes down to identifying all tuples (r_i, s_j) from the product of the two data sets $(R \times S)$ that either represent the same entity or are considered similar by custom conventions. To achieve this, a similarity metric *sim* is used to calculate a score that indicates how likely two entries represent the same entity:

$$sim(r_i, s_j) \to v, v \in [0, 1] \tag{3.1}$$

We can select a threshold $t \in [0, 1]$ to determine whether a scored tuple is a match or not. A tuple is considered a *non-match* if its score is less than the threshold. Conversely, if a tuple's score is greater than or equal to the threshold the tuple is a (possible) *match*.

Additional nomenclature:

- *Self-join*: One data set is used for input (e.g. $R = S$). The record linkage process finds duplicates in the data set. Afterwards, the identified duplicate entries can be merged or partially removed to improve the overall quality of the data set.
- *RS-join*: The collections R, S come from different sources. The structure and types of the entries may not necessarily be the the same. An RS-join is used to link entries from both data sets that represent the same entity.

The process of record linkage features three main steps: data preparation and pre-grouping of entries that are likely to share similarity, followed by scoring and matching [10, 33]. First, data needs to be standardised and transformed to a comparable format. Then, entries are scored by calculating a similarity score on their fields.

The single biggest problem, however, is the problem of complexity. No matter how quick *sim* is, it still has to be applied $|R| \times |S|$ times. As we expect only a small number of entry pairs to match, most comparisons will be made unnecessarily.

To tackle this problem and improve efficiency the input can be split into smaller sets, in a process known as *blocking*. Similarity scores are then calculated between entries within a block, which reduces the search space dramatically. Normalisation and blocking may happen immediately after entries are created or updated in the database.

3.2 Scoring and Matching of Entries

In order to make meaningful comparisons between entries we must first select suitable fields to compare against and then find a threshold for matching. These choices are typically domain-specific.

3.2.1 Similarity Metrics

To determine whether an entry pair represents the same entity, a number of their attributes are used for scoring. Typically, the fields contain string data, for instance, person names, addresses, or URLs, as a great number of techniques for approximate string matching exist [25]. For meaningful results, the fields must allow us to differentiate between distinct entities. Usually, the score is normalised to the range $[0, 1]$, with a score closer to the upper bound indicating greater similarity. Choosing a good threshold for the matching step is a challenging task. Is the threshold too high, entries that do match will be missed (*false negatives*), while a too low threshold will increase the number of incorrectly detected matches (*false positives*).

In general, similarity metrics differ by the type of input (characters, tokens or vectors), speed and accuracy or how well variations in spelling are handled. In token-based similarity metrics, the chosen tokenisation method will directly affect performance.

Several surveys have concluded that there is no universally best metric for measuring string similarity [4, 25, 2] but performance depends on the individual data. In order to improve matching quality, the results of multiple metrics can be combined to form a final decision:

$$sim(a_i, b_j) = sim_1() \geq t_1 \wedge sim_2() \geq t_2 \wedge \dots \tag{3.2}$$

3.2.2 Tokenisation and Vectorisation of Strings

A string can also be seen as a set of tokens. Hence, token-based similarity metrics count token frequencies or mark the presence of tokens in the compared strings. A simple way of tokenising a string is to split it into words using whitespace. Two problems arise from this: (1) the position of a word in a sentence is lost and (2) the strings might not contain any whitespace. The following two examples illustrate these concerns.

Example 1: The strings "alice likes bob" and "bob likes alice" are split into words.

$$\begin{array}{lcl} \mathit{split}(\text{alice likes bob}) & \rightarrow & \{\text{ alice, bob, like }\} \\ \mathit{split}(\text{bob likes alice}) & \rightarrow & \{\text{ alice, bob, likes }\} \end{array}$$

Just by examining the resulting token sets, both strings appear to be equal, even though they differ in meaning.

Example 2: The strings "apple" and "apples" are split into words on whitespace.

$$\begin{array}{lcl} \mathit{split}(\text{apple}) & \rightarrow & \{\text{ apple }\} \\ \mathit{split}(\text{apples}) & \rightarrow & \{\text{ apples }\} \end{array}$$

Both strings are very similar, in fact the only difference is one character. However, the intersection of the token sets is empty.

To attenuate these problems, *n-grams* offer a solution by splitting a string into a sequence of overlapping substrings of length n. As many algorithms for scoring as well as techniques for blocking rely on n-grams (e.g. [36]), we will introduce n-grams below.

Another interesting technique often used in information retrieval is the *vector space model*, which allows us to map a string to a vector in an m-dimensional Euclidean space

[28]. Similarity can then be calculated through algebraic operations, for instance, by measuring how far two vectors lie apart or by calculating the cosine of the enclosed angle.

N-grams

N-grams are short, overlapping substrings of length n. A string is decomposed into n-grams by sliding a fixed-length window of length n over the string. For instance, the bi-grams (or 2-grams) for “apple” and “apples” from the previous example are:

$$\begin{array}{lcl} grams(\text{apple}) & \rightarrow & \{\text{ ap, pp, pl, le }\} \\ grams(\text{apples}) & \rightarrow & \{\text{ ap, pp, pl, le, es }\} \end{array}$$

It is easy to see that this simple decomposition method allows to make a much better statement about their similarity than before. Position and frequency are, however, still lost when each individual n-gram is only registered once. In addition, the gram length n influences performance and accuracy. A larger n means fewer n-grams to store and to compare, while smaller grams result in a much greater list size.

Vector Space Model

The vector space model allows us to map an object to a vector. In information retrieval, it is often used to measure similarity between documents, but can also be applied to strings.

Each document is mapped to a vector $v_d = (t_0, \ldots, t_m)$ that acts as a fingerprint. To compare documents, vector operations can be used, such as measuring the angle between two vectors, or computing their Euclidean distance. In a basic variant of the vector space model, the vectors simply reflect the frequencies of terms in the documents. The value of each dimension is set as the number of times that a term occurs. For this, the complete set of distinct terms from all documents has to be created first. Afterwards, all documents are indexed.

Example: We assemble the frequency vectors for “apple” and “apples”. As terms, the bigrams from 3.2.2 are used. The set of distinct bigrams is { ap, es, le, pl, pp }. If a bigram occurs in the string, the value of its corresponding dimension is incremented by one.

$$\begin{array}{lcl} vectorise(\text{apple}) & \rightarrow & (1,0,1,1,1) \\ vectorise(\text{apples}) & \rightarrow & (1,1,1,1,1) \end{array}$$

Very common words that do not add information can be removed or ignored for tokenisation ("stop words") to prevent high-dimensionality of the resulting vectors. Also, to prevent bias from longer documents the vectors can be normalised by dividing each value with the number of tokens.

3.3 Normalisation

Even if data from two sources represents the same real-world objects, its structure and format often differs. For instance, the same information may be spread out over several fields in one data set, while the other combines all values in one field.

The aim of a normalisation step is to improve data quality to facilitate later comparisons [10, 29]. A number of transformations are applied, for instance, lowercasing strings, splitting or merging fields, or replacing or removing substrings.

Prior to linking entries we must first determine the fields which best identify our entities. This requires an understanding of the nature of the data, as the right amount of normalisation is highly depended on the use case. For example, stripping too many substrings will lose meaning. If the normalisation is applied too cautiously, possible matches will be missed.

For illustration, we normalised the demonstration data from table 3.1 (see table 3.2). We only selected and normalised those fields that are later used for blocking and scoring. Their values were lowercased, and excess whitespace as well as some substrings were removed. Fields were renamed. Later, after matching tuples (id_a, id_b) have been identified, the original entries can be retrieved from the data sources in table 3.1. The intermediary data sets then are no longer needed.

3.4 Blocking

So far, we have looked at how similarity can be determined by calculating a score, and how entries are prepared for matching through normalisation. What has not been discussed is how candidates for comparison are chosen. A naive way of finding all matches is scoring all possible entry pairs from data sets R and S in a nested-loop approach. A large number of comparisons $(|R| \times |S|)$ will have to be made. As we expect only a small number of entry pairs to match, this quickly becomes impractical in terms of processing time and efficiency [10, 29].

Blocking is introduced as another preprocessing step after normalisation, in order to counter growing complexity. It aims to reduce the search space for each entry to a small set of candidates that are likely to share a certain similarity. The data is grouped by sorting on fields or by using a cheap similarity metric or indexing function (as an

id_a	artistname	tracktitle
1	arcade fire	une annee sans lumiere
2	jimi hendrix	all along the watchtower
3	radiohead	electioneering
4	bob dylan	like a rolling stone
5	david bowie	life on mars
6	jimi hendrix	the wind cries mary

(a) Local music catalogue.

id_b	artistname	tracktitle
1	beatles	day tripper
2	clash the	should i stay or should i go
3	radiohead	electioneering
4	bob dylan	like a rolling stone
5	david bowie	life on mars

(b) External music catalogue that needs to be integrated.

Table 3.2: Normalised data sources that only contain the fields used for computing similarity scores. Artist names and track titles were normalised.

example, we present the *inverted index* in 3.4.1). During query time, a block with similar properties is determined. Only entries from this block are used for the more costly computation of similarity scores. Still, the quality of the overall linking is maintained. Multiple fields can be used for more precise blocking results. As with normalisation, choosing a suitable blocking strategy requires a good knowledge of the data's structure.

Table 3.3 shows the blocked music catalogue from table 3.2a after its entries were grouped by artist name.

3.4.1 Inverted Index

Inverted indexes allow for fast lookups of strings by mapping tokens to lists of string IDs. For indexing a data set, the strings are first decomposed in tokens (e.g. words or n-grams), and paired with the ID of the entry. All resulting tuples $(id, token)$ are sorted into lists. For each unique token, the inverted index stores a list of IDs.

As an example, table 3.4 shows a small set of strings and its corresponding inverted index of bigrams. For two strings to be similar, they need to share a certain number

id	artistname	tracktitle	group
1	arcade fire	une annee sans lumiere	$\circ$
4	bob dylan	like a rolling stone	$*$
5	david bowie	life on mars	$\bowtie$
2	jimi hendrix	all along the watchtower	$\triangle$
6	jimi hendrix	the wind cries mary	$\triangle$
3	radiohead	electioneering	$\star$

Table 3.3: Blocked data. Normalised entries from the local music catalogue were grouped by artist name.

of common tokens. When a lookup is performed, the search string is split into tokens. The index is then queried for each token and the resulting lists of candidate IDs are intersected.

$$query(\text{mango}) = \{2\} \cap \{1,1,2,4\} \cap \{2,4\} \cap \{2\} = \{2\}$$

Creating or updating the index can happen directly when new data is inserted into the database.

3.5 Summary

In this chapter, we have introduced the record linkage process with all involved steps: normalisation, scoring and matching, and blocking. We have found that the outcome of normalisation and performance of similarity metrics depend on the actual data. We have also indicated the problem of complexity that can be solved with a good blocking strategy.

This background will help us to choose suitable techniques for each step in the record linkage process in Chapter 4, which will then be evaluated afterwards in Chapter 5.

id	string
1	banana
2	mango
3	apple
4	orange

(a) String data

bigrams	ids
ba	1
an	1, 1, 2, 4
na	1, 1
ma	2
ng	2, 4
go	2
ap	3
pp	3
pl	3
le	3
or	4
ra	4
ge	4

(b) Inverted index

Table 3.4: Strings and inverted index of bigrams.

4 Realisation

A successful matching strategy depends on good knowledge of the expected input data. Its structure needs to be examined and suitable fields for scoring have to be identified. Then, adequate methods for normalisation, blocking and scoring as well as a threshold for determining matches can be selected.

In this chapter we will describe our efforts to choose interesting and promising techniques for each step of the record linkage process. We will also outline the preparation for the following evaluation in Chapter 5, including the creation of two data sets and the implementation a system for testing.

4.1 A Deeper Analysis of the Expected Data

First, a better understanding of structure, dimensions and properties of the input data is needed. These insights will be useful for choosing methods and techniques for the record linkage process accordingly.

We began our analysis by obtaining current data sets that were delivered by Last.fm's affiliates. We also extracted a large number of popular songs from the music catalogue. Both data sets were stored as text files with fields separated by tab stops.

4.1.1 Fields for Scoring

All data sets we examined had separate fields for artist name and track title. For scoring, we will concentrate on these two fields. To reference individual entries, the music catalogue features a unique ID field. The same is not true for any data set from the affiliates, as none offered an ID field. Preprocessing needs to take care of this fact and add a unique value field to each entry.

Additional Fields Present in the Data Sets

The data sets had additional fields that were not present in other sets. An important aspect of online music streaming is the rights management. For example, a rights holder (i.e. record labels or collecting societies) may grant playback rights for a song only for paying subscribers of a service or might limit the playback to users from certain countries. This was reflected in some data sets by extra fields that list the country codes from which a song can be accessed.

Metadata helps to improve the quality of the matching. At best, a key relationship allows us to directly link entries or a unique identifier in each data set permits distinction between entities. Two standardised codes exist to tell products and musical recordings apart, *ISRC*[1] and *UPC*[2].

- ISRC (International Standard Recording Code) is a code 12 characters long, similar to the ISBN for books. As several versions of a song can exist (e.g. studio, or live versions), it identifies a particular recording, not a song itself.
- UPC (Universal Product Code) is a unique code number for releases. It is present on many products as a machine-readable barcode.

We found fields for UPC and ISRC only in two affiliate data sets. A closer look revealed their values to be inconsistent, for instance, we saw same ISRCs for different songs. Even though the music catalogue has an ISRC column, the fields are sparsely populated. For these reasons, we did not consider ISRC and UPC any further.

4.1.2 Malformed Data

The TSV file format has no constraints for ensuring data sanity. Errors that occur during data export may alter the order of fields or produce missing values. When we examined all data sets for missing or empty fields, however, we only found an insignificant number of malformed entries. This did not include a test for lexical correctness.

4.2 Selection and Implementation of Techniques

Identifying similarities between entries is still an open and active research field [33, 10]. Many techniques for record linkage introduced in more recent publications tie the

[1]`http://musicbrainz.org/doc/ISRC`
[2]`http://musicbrainz.org/doc/Barcode`

individual steps of the process closer together and often improve how candidates for comparison are found. For instance, spatial indexes place entries in a high-dimensional Euclidean space. When a new entry is inserted, it will be placed in close proximity to similar entries, thus making it easy to select candidates [22].

With regard to the later adaptation to MapReduce in Chapter 6, we focused on easy-to-understand and well-known techniques. This is due to the fact that the MapReduce programming model allows us to distribute data processing over a cluster of machines where each node works on a small aspect of the problem. Communication between nodes, for example, to exchange intermediary results, is not intended.

We implemented all methods in Java. As there is lots of detailed documentation available for each method (e.g. publications and other implementations), we did not have to solve many noteworthy engineering issues. In the following sections we will therefore keep mentioning programming aspects to a minimum and focus more on the presentation of the algorithms.

4.2.1 Normalisation

Entries from the music catalogue and the affiliate data sets all follow their own format. Normalisation includes all transformations necessary for preparing these data sets in order to generate meaningful results in the blocking and scoring steps.

Parsing Raw Lines from a Data Source

Data sets are stored in TSV files that are read line by line. The current line is split into fields on a delimiter (i.e. tab stop), and values for artist name, track title, URL, and (if present) ID are selected. Additional fields are discarded. We treat a line as malformed when any of the necessary fields is missing or empty. Such lines are logged and not considered for further processing.

For easier handling, parsed entries are wrapped in custom data types. If data comes from an affiliate, the needed unique key is added at this point.

String Simplification

After the data exists in an independent format, several string modifiers are applied to the artist name and track title:

1. The strings are lowercased. Afterwards, we avoid a possible problem with the way strings are kept in Java. The Unicode standard allows characters to have the same appearance in spite of different binary representations. We therefore

transform strings to *Normalization Form C* (NFC), which assures that all characters are represented in a unique format [8]. Java's `Normalizer`[3] class offers this functionality.

2. Single characters are either removed or replaced. For example, curly and square brackets are changed to round brackets. Dashes, full stops and colons are replaced with whitespace. Single quotes are removed.
3. Regular expressions are used to remove longer identifiers that are often part of track titles, e.g. "studio version", "remastered", or "clean".

We used the string modifiers from the current ingestion projects as a basis but extended the modifiers for character replacements and regular expressions. For this, we analysed our data sets for frequent substrings that can be removed without altering an entry's meaning.

4.2.2 Scoring and Matching

For the selection of similarity metrics to implement and to evaluate, we reverted to previous surveys on record linkage [4, 2, 33]. We also used the functions implemented in the libraries SecondString[4] and SimMetrics[5] both for orientation and as references for our own implementations. Furthermore, for current research results, we used the extensive list of publications by the Flamingo[6] research group as a starting point.

Criteria for selection were that the techniques can operate with our data and that they are easy to understand and implement. A similarity metric needs to support approximate string comparisons, which means, it must be able to make a decision on similarity even when characters were inserted, deleted or substituted. Simple techniques are preferred over complex ones. Disk space is not as valuable as computing time and memory usage. When two similarity metrics differed only slightly in functioning (e.g. Dice's similarity coefficient and the Jaccard coefficient) or built upon another metric (e.g. Jaro distance and Jaro-Winkler distance), we only chose one.

After researching options, we selected methods introduced by Levenshtein, Jaro as well as Dice, and the cosine similarity for a thorough evaluation.

[3]`http://download.oracle.com/javase/6/docs/api/java/text/Normalizer.html`
[4]`http://secondstring.sourceforge.net/`
[5]`http://sourceforge.net/projects/simmetrics/`
[6]`http://flamingo.ics.uci.edu/`

Levenshtein Similarity Metric

The Levenshtein distance $\Delta_{ld}(r, s)$ is the minimum number of single character edits that transform string r into string s. It is often used synonymous for edit distance. Edit operations are deletion, substitution and insertion. Each operation is assigned a cost. Thereby, each kind of edit may be penalised or favoured by declaring different costs.

Example: Δ_{ld}("jimmy hendri", "jimi hendrix") = 3 (with equal costs for each edit operation)

jimmy hendri	$\rightarrow$	jimm hendri	(deletion of 'y')
jimm hendri	$\rightarrow$	jimi hendri	(substitution of 'm' with 'i')
jimi hendri	$\rightarrow$	jimi hendrix	(insertion of 'x')

The calculated value still is a distance measure. We used formula 4.1 to map a distance to a similarity score $s \in [0, 1]$.

$$Levenshtein(r, s) = 1 - \frac{\Delta_{ld}(r, s)}{max(|r|, |s|)} \tag{4.1}$$

We based our implementation on code[7] published by Chas Emerick, which is supposed to be more memory efficient for longer strings.

Jaro Similarity Metric

The Jaro similarity metric for strings r and s is defined as

$$Jaro(r, s) = \frac{1}{3}\left(\frac{|m_r|}{|r|} + \frac{|m_s|}{|s|} + \frac{m_r - T_{m_r,m_s}}{m_r}\right) \tag{4.2}$$

In order to calculate a score, two things are needed: the number of *matching characters* (m_r, m_s), and half the number of *transpositions* (T_{m_r,m_s}).

- Matching characters: a character is *matching* if it is present in both strings, and its occurrences do not lay further apart than $\lfloor \frac{max(|r|,|s|)}{2} \rfloor - 1$.
- Character transpositions: similar strings have the same matching characters, their ordering in the strings, however, can be different $(a_i \neq b_i)$. Those occurrences are counted.

An implementation of the Jaro similarity metric was found in the SecondString library, which we followed for our implementation.

[7]`http://www.merriampark.com/ldjava.htm`

Dice Similarity Coefficient

At first, the input strings r, s are tokenised in sets $S_r = \{r_{t1}, \ldots, r_{tn}\}$ and $S_s = \{s_{t1}, \ldots, s_{tm}\}$. Similarity then is determined by comparing how many common tokens make up the strings.

$$Dice(r, s) = \frac{2|S_r \cap S_s|}{|S_r| + |S_s|} \tag{4.3}$$

Java's `HashSet` only holds unique elements. Therefore, our implementation of this metric can give imprecise results with strings that have repeating sequences. For example, the bigram sets for "aa" and "aaaaaa" are equal. A variation would be to produce a "bag" of tokens, rather than a set, so each token occurrence would appear in the result. To facilitate implementation, however, we did not use bags of tokens.

Cosine Similarity

The cosine of the angle between two vectors is suitable for stating similarity. When strings are mapped to term-frequency vectors, the angle between two vectors will always be between 0° and 90°, since term-frequencies are always positive or zero. As a result, the cosine is always in the range $[0, 1]$.

$$Cosine(r, s) = \cos(\alpha) = \frac{v_r \cdot v_s}{\|v_r\| \|v_s\|} \tag{4.4}$$

For vectorisation, we use the string-to-vector mapping that was outlined in section 3.2.2.

4.2.3 Blocking

Apart from the already described inverted index (see 3.4.1), we were interested in the performance of a spatial index where all data is mapped to a m-dimensional Euclidean space.

Inverted index

In section 3.4, we explained how an inverted index is created and can be queried. However, if the lists of string IDs are simply intersected for the query answer, it will only be able to find exact matches.

In order to respond to approximate queries, the following fact can be used with n-grams: If all strings within edit distance k are accepted as candidates ($\Delta_e(r, s) \leq k$), then they must share at least the following number l of common n-grams with query string r [36]:

$$l = |r| + 1 - (k + 1) \cdot n \tag{4.5}$$

The set of string IDs can therefore be filtered with l as a lower-bound for the number of common n-grams.

For our experiments, the inverted index was kept in memory in a `HashTable<String, Set<Long>>`. By this, however, we could not use aforementioned formula 4.5 for defining an exact lower-bound as each occurrence of a token in a string can only be stored once.

Approximate queries, however, are possible with our implementation, although the preselection of candidates is less strict and we accept a higher number of similarity comparisons later in the process. A maximum token difference d must be provided when the inverted index is queried with r. Let $S_c = S_r \cap S_s$ be all tokens that are present in r and a candidate s ("common tokens"). Only those s that satisfy the necessary condition 4.6 are returned.

$$\max(|S_r|, |S_s|) - |S_c| \leq d \tag{4.6}$$

When the index is queried, all found IDs are grouped by the individual string ID. Conveniently, the size of each group directly is the number of common tokens.

Spatial Index

We chose the spatial index approach mainly out of curiosity as we assumed its adaptation to MapReduce to be overwhelming.

For indexing, the data is moved to an Euclidean space of dimensionality m. Then, all vectorised strings are clustered. Strings that share similarities will fall in the same clusters and can be identified with little effort. We decided against using term-frequency vectors as they display (very high) dimensionality, which is determined by the number of all distinct tokens occurring in the data set.

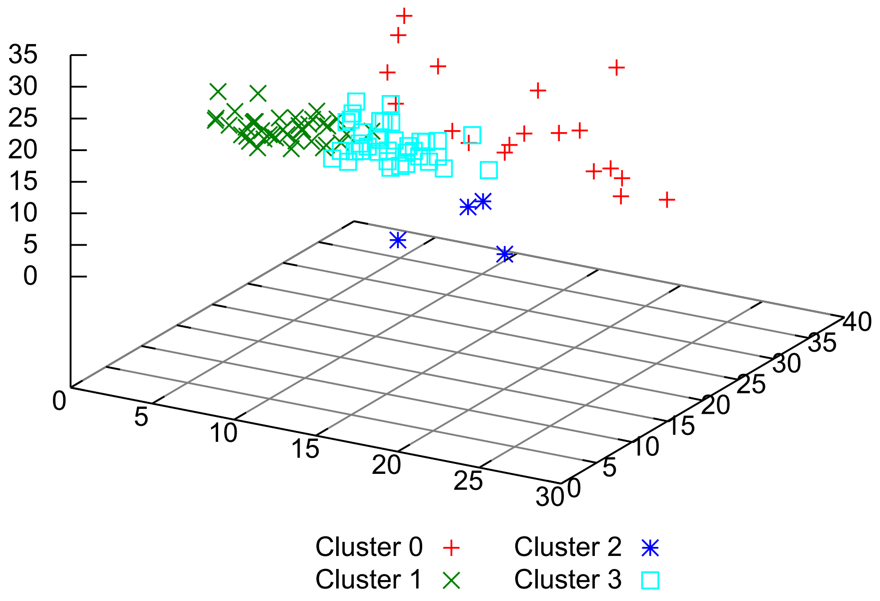


Figure 4.1: Example classification of 100 strings in four clusters in a 3-dimensional Euclidean space.

This left us in search for two main components, a technique for mapping strings to vectors, and a simple clustering algorithm. We finally chose *StringMap* [22] as string-to-vector mapper, and the popular *k-means clustering* algorithm [23] for grouping vectors.

Building the spatial index comes down to four steps:

1. Initialing StringMap with a small training set of strings.
2. Mapping the whole data set to the defined m-dimensional Euclidean space.
3. Training the k-means algorithm with the mapped training data set to initialise the center points (*centroids*) of its k clusters.
4. Classifying all vectors. Store in an extra field for each string what cluster it is assigned to.

Now, for finding similar entries, the query string is also mapped to a vector and classified afterwards. All entries with the same cluster ID are selected from the data set as possible candidates. Pictured in figure 4.1 is an example mapping and successive clustering of 100 strings. Below, we will discuss aspects of our realisation, for example, some details on the implementation, modifications we made or general functionality.

StringMap StringMap is a method that allows a transformation of strings to an m-dimensional Euclidean space. The transformation is distance-preserving, that means, the positions of points in the Euclidean space reflect the similarity of their original strings. Also notable is that the dimensionality m is a user-defined parameter. The problem of high-dimensionality from other mapping methods, e.g. term-frequency counting, is thereby averted.

In an initialisation phase the Euclidean space is constructed. For this, we use a random subset of strings and not the whole data set. The quality of the mapping may therefore by influenced by the strings that are used for initialisation.

In their paper on StringMap, Liang et al. first transform all entries of the data set to a Euclidean space and then insert the resulting vector into an *R-Tree* (a spatial data structure) to find similar objects. For a good dimensionality m, they advise $m = 20$, which we followed. However, for our spatial index approach we used StringMap exclusively for mapping strings to the Euclidean space.

We reimplemented the reference implementation[8] (C++) by Liang and Chen in Java. In order to further optimise performance, we made minor modifications to the structure of the program code.

K-means Clustering Clustering is a method for partitioning data in subsets (*clusters*) so that elements in the same cluster share similar attributes. k-means clustering works with a fixed number of clusters. The centre points of clusters are called centroids.

As mentioned, StringMap preserves distances. We can therefore use the Euclidean distance to find the nearest centroid $c = (c_1, \dots, c_m)$ for a given point $v = (v_1, \dots, v_m)$:

$$d(v, c) = \sqrt{\sum_{i+1}^{m} (v_i - c_i)^2} \tag{4.7}$$

In the assignment step, all points are assigned to clusters. Then, the update step recalculates each centroid. The new centroids are the means of all points in the cluster. Both steps are repeated until the assignments do not change, or a certain number of iterations is reached.

Details of our implementation:

- A higher number of clusters allows a finer distinction between entries. However, it also increases the complexity for finding a cluster. For this reason, we set the number k of clusters to $k = \lfloor \frac{1}{3} \cdot |S_{\text{training Vectors}}| \rfloor$.

[8]`http://flamingo.ics.uci.edu/releases/4.0/`

- The *Random Partition* method is applied to initialise the centroids. Here, each vector is first assigned to a cluster at random, then centroids are updated.
- The mapped training vectors from the earlier StringMap initialisation are used for finding centroids.

4.3 Preparation for Evaluation

In preparation for the evaluation in the following Chapter 5, we had to implement a prototype system that allowed us easily test different methods and measure their performance. As we plan to identify a combination of techniques that performs best for data used at Last.fm, we also created own data sets for testing.

4.3.1 Implementation of a Prototype System

For evaluation we wrote a prototype system called `affiligator` that allowed us to easily plug-in different techniques for each step of the record linkage process. It also allowed us to easily read and write files, as well as to measure values like running times or number of matches found.

We used the Guava Libraries[9] from Google as well as Commons IO[10] from Apache Commons to facilitate development. These libraries offer a ready collection of functionality that would otherwise have to be expressed with plain Java language constructs, for example, reading all lines from a file directly to a list or intersecting two sets.

4.3.2 Creating Test Data Sets

We did not use any data sets commonly used for evaluation in other research publications, for example, lists of actors names or movie titles from the Internet Movie Database[11].

These data sets are very useful for three reasons: (1) they are publicly available and easy to obtain, (2) their character is known and (3) they allow a comparison of results of new approaches and techniques between research teams. However, as we had been looking for a solution that performs best for our prerequisites, we opted for creating our own data sets from real-world data instead.

[9]`http://code.google.com/p/guava-libraries/`
[10]`http://commons.apache.org/io/`
[11]`http://www.imdb.com/interfaces`

Data set	String #	String length.		
		Min	Max	Avg
`popcat`	50,000	5	212	27.27
`randaff`	11,777	9	298	45.74

Table 4.1: Descriptions of the data sets used for evaluation.

Starting points were two files, a recent delivery from an affiliate and an excerpt of the most popular songs from the music catalogue. Both sets were very large and deemed impractical for constant use. To make working with the data more manageable, we sorted the music catalogue set descending by popularity and selected the top 50000 entries. From the affiliate file, we randomly selected about 11000 entries. In the following, the new data sets will be referred to as `popcat` (music catalogue) and `randaff` (affiliate data).

In table 4.1 the minimum, maximum and average lengths of strings (artist name and track title combined) for each data set are listed. We were surprised by the maximum string lengths. A short investigation, however, revealed no inconsistencies in the sets, but a small number of "overly-descriptive" track titles. We did not alter the data sets any further.

4.4 Summary

Over the course of this chapter we have selected several methods for each step of the record linkage process. These include techniques for determining similarity of entry pairs, normalising strings and finding candidate strings by grouping similar entries from a data set.

In order to identify techniques that are best suited for demands at Last.fm, we have created our own data sets. We have also implemented a system that will allow us to evaluate each methods individually in the next Chapter 5.

5 Evaluation and Experiments

In this chapter, we will evaluate the methods we have selected in Chapter 4, and identify several combinations of methods that perform best for our demands.

We begin by evaluating each step of the record linkage process individually. We score similarity metrics by measuring running times, and precision and recall. As the nature of our blocking techniques is very different, we compare both approaches by how complex it is to index data sets and how accurately the index can be queried. In order to make a statement on normalisation, we test how string simplifiers affect the number of duplicates in a data set. Individual experiments make it necessary to derive additional data sets from our `popcat` and `randaff` data sets.

Best performing methods are finally combined, and overall performance is determined.

5.1 Test Setup

All of our experiments and tests were run on a standard desktop computer with 4GB of RAM, and an Intel Pentium D 2140 dual-core CPU running at 1.60GHz. The operating system was Ubuntu Linux 10.10 (64-bit).

As described in Chapter 4, our methods were implemented in Java, which was installed in version 1.6.0_24-b07. The data sets were read to memory before any measurements were conducted.

5.2 Normalisation

We extracted artist names and track titles from our `popcat` and `randaff` data sets, and saved the resulting 61777 strings in a new file. Our experiment was based on the assumption that the file contains duplicates and that normalisation will increase their number. This will later translate to a higher chance of finding equal matches between two data sets.

Successively, all string simplifiers were applied. After each simplifier, a snapshot of the data was saved for inspection. Table 5.1 lists the effects of the normalisation step. We

String simplifier	Duplicate #	$\Delta_{duplicates}$	Avg. string length.
—	331	—	30.789
Normalisation to NFC	331	0	30.789
Lowercasing strings	353	22	30.789
Single-character substitution	445	92	30.736
Regular-expression substitution	458	13	30.617
Removal of whitespace	564	106	30.460

Table 5.1: Duplicates and average string lengths after each string simplifier was applied.

saw no noteworthy changes in average string length but the number of duplicates rose by 233.

5.3 Scoring and Matching

We aimed to find a fast similarity metric and an appropriate threshold that allow us to reliably decide whether an entry pair is a match or non-match.

In order to decide this, we focused on (1) efficiency of the comparison and (2) precision and recall for various thresholds. For the second experiment, two additional data sets were needed, where the expected number of exact and approximate matches was known, but which could also provoke false-positive results.

5.3.1 Efficiency of one Similarity Comparison

Whether the compared entries are later deemed similar or dissimilar, the cost for their comparison is same. Therefore, to test efficiency, we carried out a large number of comparisons and measured the overall time needed to complete the task (in nanoseconds). To even out deviations, we repeated each test ten times and discarded the highest measurement. The remaining values were used to calculate how long a similarity metric needed on average to score an entry pair.

In table 5.2, we list results per similarity metric and for different input formats. Cosine similarity achieved best performance, closely followed by the Dice similarity coefficient. Times were higher when the compared strings had to be split in n-grams or mapped to a vector prior to comparison.

Although differences in speed do exist, we find it questionable if those are relevant in later production use. For example, the difference between the fastest metric (cosine similarity with words) and the slowest metric (cosine similarity with trigrams) is about

Similarity metric	Input format		
	words	bigrams	trigrams
Levenshtein similarity metric	9.8391	–	–
Jaro similarity metric	7.8638	–	–
Dice similarity coefficient	5.1135	15.8929	15.9929
Cosine similarity	4.9639	28.1271	30.4043

Table 5.2: Averaged running times for one comparison in μs.

26 μs for one comparison. For a noticeable time difference of ten minutes, 23 million comparisons have to be made. We did not consider running times in the later recommendation.

5.3.2 Precision and Recall

How accurate the results of a matching are depends not only on the similarity metric but is also influenced by the chosen threshold. We can divide results in exact matches, (correct) approximate matches, false positives, and false negatives.

Exact matches have a score of 1.0 and can easily be identified. Approximate matches and false positives are more difficult to tell apart, as both have scores in the same scoring range. Results will have to be classified manually. False negatives (i.e. missed matches) can only be stated when it is known how many correct matches should have been returned. A proper test of precision and recall therefore demands data sets where all matches have been identified and classified before.

Derivation of Data Sets

We created two intermediary data sets by sampling 100 random artist names from `popcat`, and selecting up to ten of their songs from `randaff` (with utility `grep`). We proceeded to identify matches.

1. Both data sets were matched by our `affiligator` prototype. We set a low threshold of $t = 0.55$ and used Levenshtein's similarity metric. This produced a large number of matches.
2. All matches were manually classified in correct matches, and false positives. We considered a match correct not when it had a high score but when it met our expectations.

3. The list of correct matches was used to select source entries from `popcat` and from `randaff`. Duplicate entries were removed from the resulting lists. Artist names and track titles were normalised.

These derived data sets had a small number of entries and contained 303 exact matches and 42 approximate matches.

F-Scores

F-scores allow for easy comparison of results by calculating a score that measures the ratio between correct matches, false positives, and missed matches. For this, precision and recall are weighted. A score closer to 1.0 denotes more correct results and less false positives.

The general formula for calculating F-scores is

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{5.1}$$

with

$$\text{precision} = \frac{\text{number of correct results}}{\text{number of all results}} \tag{5.2}$$

and

$$\text{recall} = \frac{\text{number of correct results}}{\text{number of expected results}} \tag{5.3}$$

The F_1 score is often used, which equally weighs precision and recall. We, however, were more interested in $\beta = 0.5$ and $\beta = 2.0$.

Higher precision means more correct matches and less false positives but also a smaller match rate. Of course, there is a chance that matches are missed. The $F_{0.5}$ score reflects wish, and is biased towards more accurate results. In contrast, a commercial point of view might regard a higher match rate as more valuable than precision. For instance, instead of not displaying any link at all, a user could still be presented a “not quite what was asked for, but possibly interesting” result that directs the user to an affiliate’s service. The F_2 score therefore favours recall over precision.

For this test, we scored all possible entry pairs from our testing data sets and analysed matches. We found that all similarity metrics reliably identified exact matches. Hence, we did not consider exact matches for calculating F-Scores but only the results from approximate matching. We also learned that false positives are due to strings with long

Similarity metric	F-score, threshold and input type.					
	$F_{0.5}$			F_2		
Levenshtein similarity metric	0.5851	0.9	words	0.3039	0.9	words
Jaro similarity metric	0.3667	0.95	words	0.4321	0.9	words
Dice similarity coefficient	0.5159	0.9	2-grams	0.5834	0.8	3-grams
Cosine similarity	0.5000	0.90	2-grams	0.5837	0.8	3-grams

Table 5.3: F-scores with thresholds and input types for the tested similarity metrics.

matching portions, while the difference that changes meaning is relatively small, e.g. "ro*ll*ing" and "ro*tt*ing".

The best performing combinations of input type and threshold with resulting $F_{0.5}$ and F_2 scores are listed in table 5.3. Furthermore, the graphs shown in figure 5.1 demonstrate how both scores develop for different thresholds. Overall, the Dice similarity coefficient performed well in both cases with cosine similarity having a marginally better F_2 score. The Levenshtein similarity metric achieved best results for accuracy.

5.4 Blocking

The nature of the evaluated blocking techniques is very different. In order to compare the spatial index with the inverted index, we focused our experiments on the costs for indexing data sets and how accurate and fast the index can be queried.

Our measurements covered:

- Average times for indexing one entry. We also looked at how many string IDs/-clusters were created and how these were populated.
- Average times for querying an index for similar entries and how many strings were returned with an answer.

We used the entries from `popcat` for creating the indexes as well as for querying. Thereby, we made sure that each query returned at least one result. In order to get a feeling for performance, we repeated the experiments with various entry numbers (50, 500, 5000, and 50000).

Token type	Indexing one entry (μs)	Lists of string ids		
		$\sum$	$\min_{ids}$	$\max_{ids}$
2-grams	10.148	2,167	1	2,6456
3-grams	17.768	16,522	1	10,242
5-grams	20.547	18,4039	1	6,269
7-grams	28.556	467,075	1	1,342
words	5.439	32,711	1	8,710

(a) Indexing

Token type	Querying one entry (μs)	String ids returned		
		avg_{ids}	$\min_{ids}$	$\max_{ids}$
2-grams	93,683.855	1.22972	1	21
3-grams	20,871.722	1.13928	1	20
5-grams	1,991.730	1.07696	1	17
7-grams	417.170	1.0578	1	17
words	2,406.132	523.22024	1	5,914

(b) Querying

Table 5.4: Results for creating and querying an inverted index (50000 entries, upper bound for token difference was $d = 4$).

5.4.1 Inverted Index

The inverted index was tested with different n-grams and words as tokens. We allowed token differences from a strict $d = 0$ (only exact results) to a more lax $d = 5$ (approximate answers) for querying. We present an exemplary overview of our measurements for 50000 entries and $d = 4$ in table 5.4.

An inverted index with short n-grams consumes less space but query times will be higher. The opposite is true for longer n-grams. When we performed our tests with words as tokens, indexing was fast but the number of string IDs that were returned on average was significantly higher than with any gram length. We saw overall good performance for 5-grams.

5.4.2 Spatial Index

We present our results in table 5.5. Our spatial index approach had shortcomings. Times for indexing as well as querying were much higher throughout than for the inverted index.

Entry #	Initialisation (μs)		Cluster #	Indexing one entry (μs)
	StringMap	k-means clustering		
50	121,805.659	2,766.357	16	1,519.956
500	408,639.616	115,744.216	166	2,219.236
5000	3,710,097.529	17,602,828.763	1,666	4,695.054
50000	30,510,175.489	585,601,778.598	16,666	24,673.817

(a) Indexing

Entry #	Querying one entry (μs)	String ids returned.		
		avg_{ids}	min_{ids}	max_{ids}
50	1,526.455	5.800	1	10
500	2,242.022	5.372	1	12
5000	4,731.014	5.824	1	25
50000	24,656.594	5.178	1	23

(b) Querying

Table 5.5: Performance of the spatial index approach for increasing data set sizes.

We attribute this to the complex components (i.e. StringMap and k-means clustering) that are involved. Before any actual indexing can commence, they must be trained. For querying, an entry must be first mapped to a string and classified. Only then can similar entries be retrieved.

The numbers for string IDs returned on average were fairly stable, possibly due to the growing number of clusters.

5.4.3 Discussion of Index Types

An inverted index trades in space for speed. All unique tokens and the list of string IDs have to be stored in addition to the indexed data set. When entries are deleted or updated, all of their references in the inverted index have to be found and updated as well.

In contrast, a spatial index will consume almost no extra space. Every indexed entry only needs to mark the cluster it belongs to. Unfortunately, the index will have to be rebuilt entirely when the character of the data changes over time. In addition, the non-deterministic aspects of StringMap and the clustering algorithm might translate to imprecise classifications (e.g. too many entries in one cluster or similar entries in different clusters). In this case, indexing should be repeated.

Combination	Blocking		Scoring		
	Token type	d	Similarity metric	Input format	t
A	5-grams	12	Levenshtein similarity metric	words	0.9
B	5-grams	20	Dice similarity coefficient	3-grams	0.8
C	5-grams	24	Cosine similarity	3-grams	0.8
D	3-grams	20	Dice similarity coefficient	3-grams	0.8

Table 5.6: The identified combinations of techniques that perform best for our data sets. All combinations use the same string simplifiers for normalisation.

The spatial index presents an interesting concept. We are, however, concerned by the measured performance of our approach and its potentially complicated adaptation to MapReduce. We therefore favoured the inverted index over the spatial index for the remainder of our work.

5.5 Combining Results and Summary

All techniques and methods were thoroughly evaluated separately. We can now identify several combinations of techniques from the individual results of our experiments. These are listed in table 5.6.

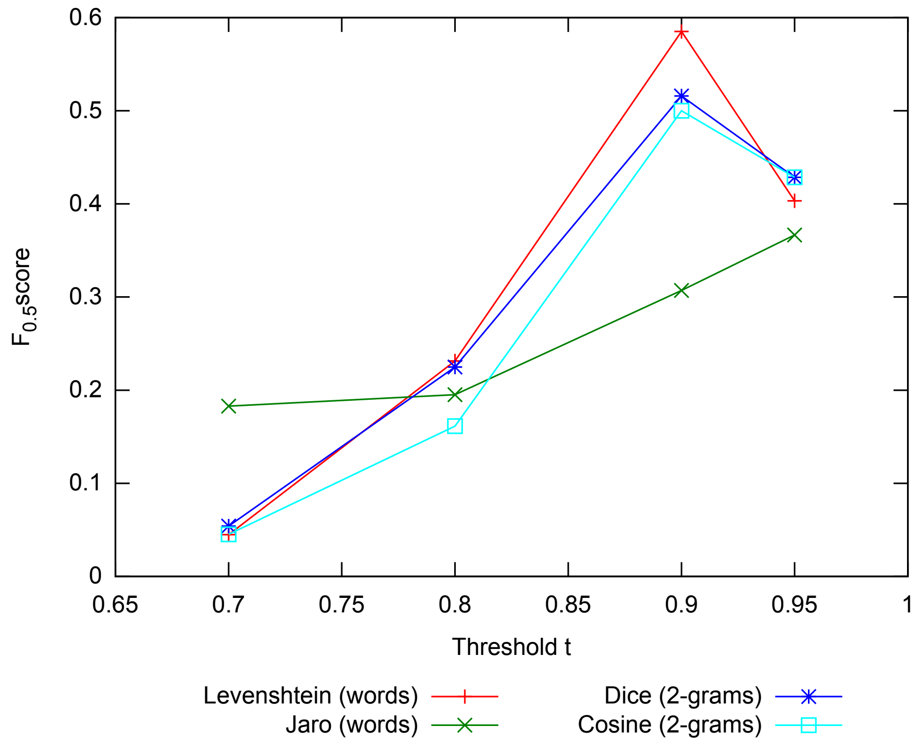
For more precise results we advise combination A, or combinations B, C or D[1] for better recall. Normalisation is same for all combinations. In order to find optimal token differences d, we reran our tests from 5.3.2, exchanged the nested-loop approach for an inverted index and tried different values for d. An optimum was found when F-scores for the results were same as previously measured (see figure 5.2 for an example).

As a final test, the combinations were used to match our data sets `popcat` and `randaff`. We divided matches in exact and approximate matches and randomly sampled 100 approximate matches for thorough examination. We list these results in table 5.7.

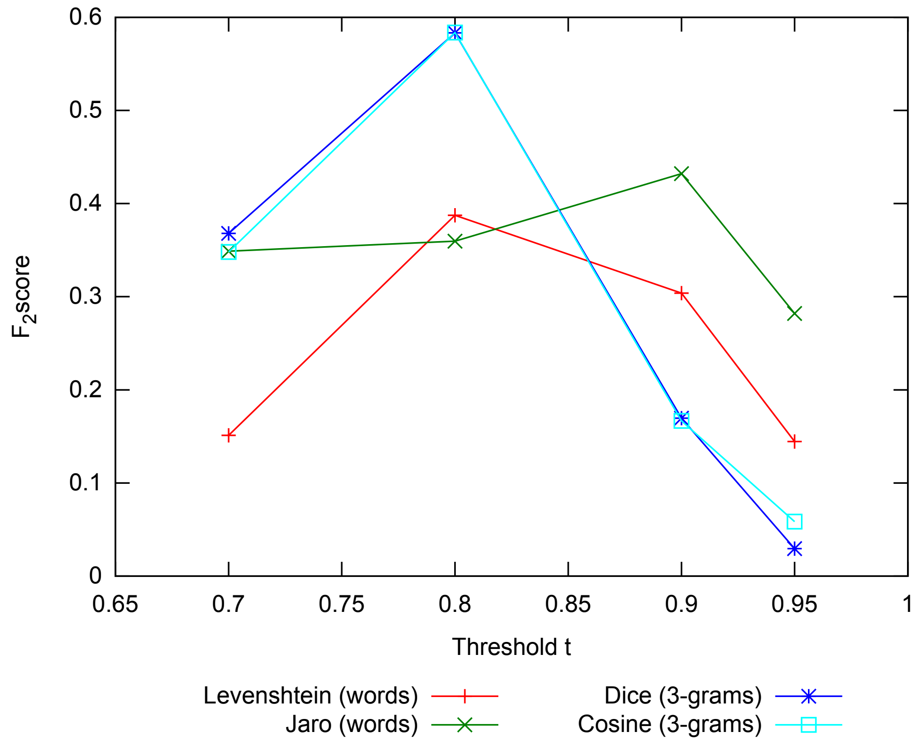
[1]Combination D was identified especially for section 6.2.4 and is not included in further measurements in this chapter.

Combination	Comparison #	Matches			Sample	
		$\sum$	Exact	Approximate	Correct	False positives
A	45,499	274	115	159	65	35
B	2,164,463	533	115	418	53	47
C	3,491,660	633	115	518	43	57

Table 5.7: Results for concluding tests. The identified combinations were used to match data sets `popcat` (50000 entries) and `randaff` (11777 entries).



(a) $F_{0.5}$ score



(b) F_2 score

Figure 5.1: Development of F-scores for the evaluated similarity metrics.

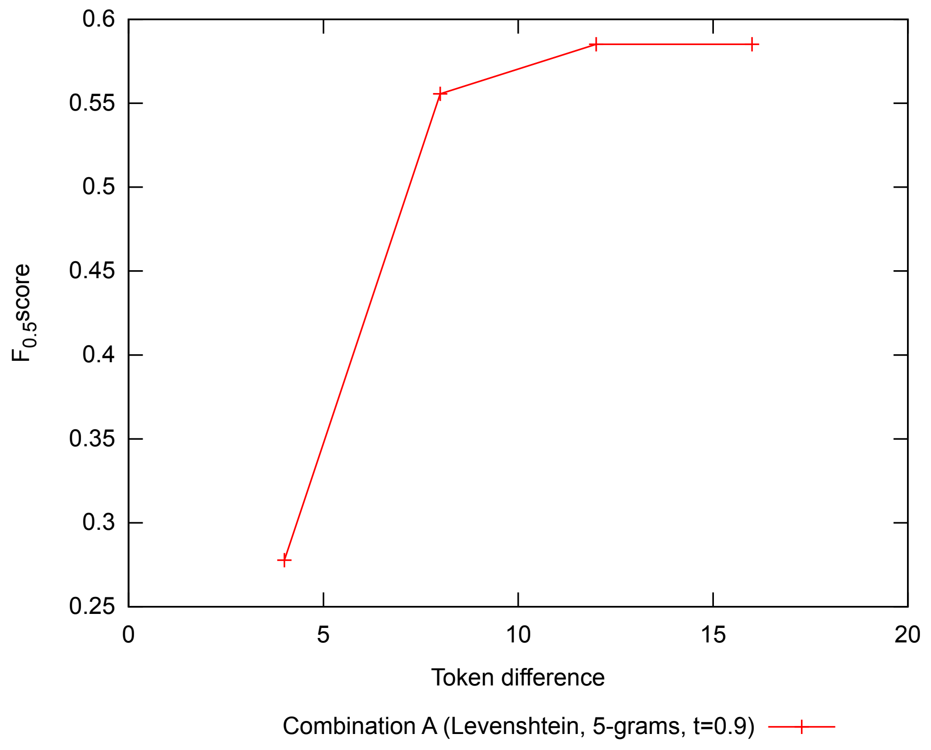


Figure 5.2: $F_{0.5}$ scores for the Levenshtein similarity metric correlated with token difference d from the inverted index. Peak performance is reached for $d = 12$.

6 Adaptation to MapReduce

In this chapter we will present our efforts to adapt the combinations of best performing techniques we identified in Chapter 5 to the MapReduce model and its open source implementation Hadoop. For faster development we choose not to implement our solution in "raw" MapReduce, but use the Cascading framework.

We will begin with an introduction of MapReduce, Hadoop, and Cascading. Next, we present notes on our adaptation. We will test our program with larger data sets on Last.fm's computing cluster, and measure performance. Finally, we summarise and discuss results, and give possible improvements for our solution.

6.1 Introduction to MapReduce, Apache Hadoop and Cascading

Even when a computational problem is well-understood, and elegant solutions to it do exist, the growing scale of data collections will add a whole new level of complexity to it. Data sets of today store millions, or billions of entries, and cannot be kept in memory on one computer. In order to process this amounts in reasonable time, processing has to be done in parallel, distributed over a number of machines.

Big computing clusters often consist of hundreds, or thousands of nodes. These have to be employed evenly and supplied with input data. Results must be collected from nodes and stored. With a growing number of components, network problems and component failure become more likely. A program would have to react to this, and redistribute the failed work units to another free node. All this demands additional effort by the programmer. The actual solution to the initial problem disappears behind extensive program code.

6.1.1 MapReduce

MapReduce [9] frees the programmer from this extra work and provides a simple way for data-centric applications to be run on a cluster of machines. Since Jeffrey Dean and Sanjay Ghemawat, both employed at Google, introduced MapReduce in 2004, it has gained widespread interest.

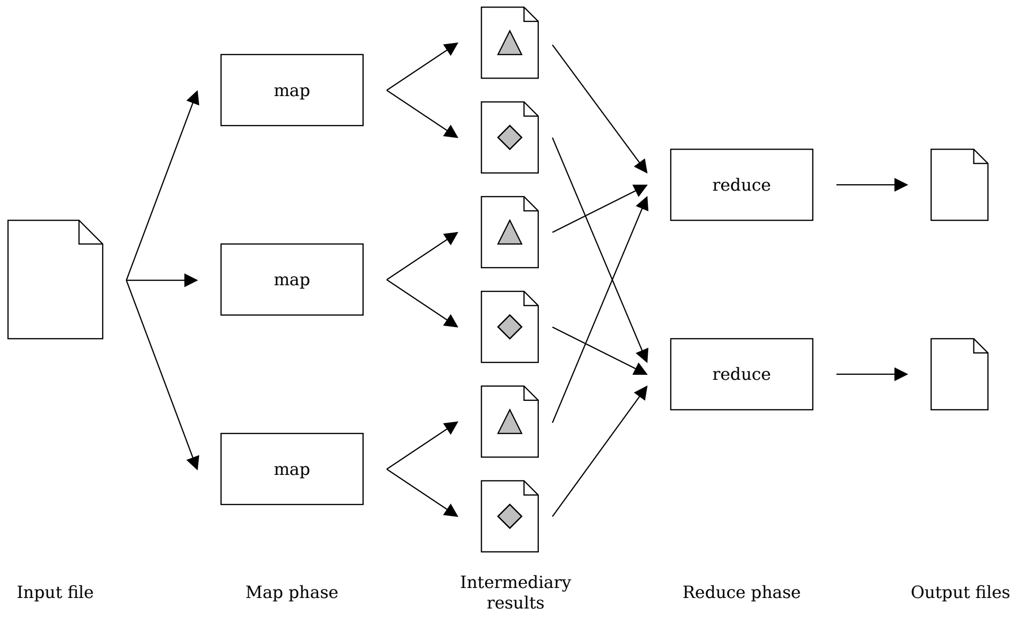


Figure 6.1: Execution overview of map and reduce phases [17].

When a computational problem can be expressed in MapReduce, it will automatically be run in parallel, and scale over a cluster of few, but also hundreds or thousands of inexpensive machines. Most hardware failures, high load situations or network problems are recognised, and taken care of. Details for parallelisation, fault tolerance, and distribution of data and work units are hidden from the programmer. The programmer can thereby concentrate more on solving the actual problem.

For this, MapReduce introduces a strict but simple programming model. This demands, for example, that each step needs to be a collection of sequential tasks. Also, nodes cannot exchange intermediary results during program execution.

Components of MapReduce

Most important for MapReduce are the *map* and *reduce* functions. These are written by the user, and operate on data (figure 6.1). A MapReduce job can have several alternating map and reduce functions; the output from a map function is used as input for a reducer.

Not only are nodes used for computation but also for data storage. Data sets are kept in a simple, semi-structured format. Prior to execution, the input data is partitioned in small parts, and parsed in $(key, value)$ pairs. Each node will only work on small aspects of the problem at a time. Intermediary results are collected, and later combined to the final result of the computation, which is again stored in plain text files. If a node fails its work unit is rescheduled to another node. As work units are small, a failure does not severely affect the overall computation.

Map The map function takes a $(key, value)$ pair and assigns it a key based on some criteria. Thus, similar tuples are assigned the same keys.

$$map(k_1, v_1) \rightarrow list(k_2, v_2) \tag{6.1}$$

As key assignment is independent from other pairs, all mappings can be done at the same time. Following, all pairs are collected and grouped on their new keys k_2. This is done very fast by a grouper that uses a large, distributed sorting algorithm.

Reduce All pairs with the key k_2 are presented to the same reducer. A reducer operates on all values in a group and returns a new list of values.

$$reduce(k_2, list(v_2)) \rightarrow list(v_3) \tag{6.2}$$

All reduce functions can also be executed concurrently, as each reducer only operates on results with the same key k_2, i.e. there exist no dependencies to other groups. All results from the reducers are either combined and form the final result of the computation or act as input for following map functions.

Additional Components Around the mentioned map and reduce functions numerous other components exist that make up MapReduce. This includes functionality for reading, writing and partitioning files in a distributed environment. There must be a way to compare, group and merge data. Much needed is also a job scheduler that assigns work units, keeps track of results, and redistributes work to free nodes in case of failures.

Distributed File Storage and Location Awareness

Network bandwidth is a scarce resource in a cluster. Moving large amounts of data between nodes causes high load and quickly maxes out the network's capacity. This has led Google to implement their own file system for distributed data storage, named *Google File System* or GFS [12].

In GFS, data is split in several chunks and stored on a number of inexpensive machines. One important feature is *locality*: The file system will not only keep track of what nodes certain chunks are kept on but also what network segments they belong to and how these segments are connected.

MapReduce relies heavily on locality to improve efficiency. Cluster nodes are not only used for computation but also for data storage, thus eliminating the need for an expensive, dedicated storage solution. As it is easier to assign work units based on locality than for a node to request data first, the job scheduler will try to minimise the amount of data that needs to be sent between nodes. At best, a node can directly perform a computation on the data it is storing. If data needs to be sent, it is tried to place the job close to the data, i.e. in the same network segment or to a node in the same rack.

For data availability and fault tolerance, each data chunk is replicated across multiple nodes when data is imported to GFS.

Applications

Due to its great popularity, MapReduce is often seen as a cure-all for every kind of data processing problem. It has found its way into existing database management systems for structured or semi-structured data, such as CouchDB[1] or Kyoto Cabinet[2]. Also, MapReduce frameworks have been implemented for a variety of programming languages.

Nevertheless, MapReduce is not a general-purpose tool. Its programming model introduces restrictions that a program has to operate in: Map and reduce functions are stateless, and cannot have data dependencies with other jobs, as those are executed on other nodes in separate runtime environments. This means that all data must be collected and grouped before it is presented to a map or function. MapReduce is meant for batch computations. Result can later be imported to a database where they get indexed for successive use.

Classic applications, mentioned in the original publication, are searching and sorting data, data mining in large log files, document clustering, and large-scale indexing [9]. Since then, MapReduce has been used to solve record linkage problems (e.g. [31, 21, 32]), but also in many other fields, for example, electronic commerce, data mining in social networks, machine translation or biology. There has also been research on how to adapt graph algorithms, and spatial problems to MapReduce (e.g. [3]). See [30] for an exemplary list of use cases and research publications.

Example

A frequently used example to explain the MapReduce model is counting word frequencies in documents [9], which we present here slightly modified. Suppose we want to learn the

[1]`http://couchdb.apache.org/`
[2]`http://fallabs.com/kyotocabinet/`

occurrences of each different n-gram in a large document. For this, a user would have to write one map, and one reduce function.

- Map: The document is first partitioned, for example in sentences; these are passed to the map function. Each word from a sentence is decomposed in n-grams, which are emitted in tuples with and initial count of 1:

$$map(\text{document, mango banana} \ldots) \rightarrow (\text{ma}, 1),\ (\text{an}, 1),\ (\text{ng}, 1),\ \ldots$$

- Reduce: Tuples are sorted and grouped. All tuples from one group are fed to the same reduce function. This function just needs to determine the size of the group to find how often an n-gram occurs in the document:

$$reduce((\text{ng}, 1),\ (\text{ng}, 1),\ (\text{ng}, 1)) \rightarrow (\text{ng}, 3)$$

Outputs from the reduce functions are collected, and make up the list of n-gram occurrences.

6.1.2 Apache Hadoop

Apache Hadoop[3] is an open source implementation of MapReduce and the Google File System written in Java. The project is organised in three sub-projects: Hadoop MapReduce, Hadoop Distributed File System, and Hadoop Common.

Programs for Hadoop are written in Java, but libraries for other programming languages also do exist [16]. Additionally, using a streaming API[4] every program, for instance, shell scripts, can be used as map or reduce functions.

File Systems

For distributed data storage, Hadoop offers the *Hadoop Distributed File System* (HDFS) [14]. It is based on the concepts of GFS, and also offers location awareness and data replication. Of course, data can also be accessed from other file systems. Knowledge of locality, however, might be lost with other file systems than HDFS.

[3] `http://hadoop.apache.org/`

[4] `http://wiki.apache.org/hadoop/HadoopStreaming`

Hadoop Distributed File System Before data can be accessed from Hadoop, it needs to be imported. Importing as well as exporting data is either done by command line tools or via an API [15]. Data will be split it in a series of blocks, with block sizes being of multiples of 64MB. By default, a block is replicated to three nodes in the cluster for data availability, with two nodes in the same rack and one node in a different network segment. Thereby, when a node fails, the data blocks can still be read from another location.

Other File Systems Besides HDFS, other file systems are available as well. For instance, data can come from any local file system the underlying operating system can access or from cloud storage service Amazon S3.

Native support for cloud services is interesting for companies that have built their applications on top of platforms like Amazon AWS[5]. In this case, their data does not need to be transferred before computation but can directly be accessed. A company can either run their own Hadoop cluster on top of cloud infrastructure "on demand", or submit MapReduce jobs to Amazon Elastic MapReduce[6], a version of Hadoop that was customised by Amazon to hide to underlying infrastructure even further.

Architecture

Hadoop and HDFS follow a master/slave architecture [18]: One node is designated *master node*, and exclusively runs the *JobTracker* and *NameNode* processes; all other nodes run *TaskTracker* and *DataNode* processes (figure 6.2).

- JobTracker and TaskTracker provide MapReduce functionality. A user program is submitted to the JobTracker, which manages execution, and distributes work units. From here, tasks are assigned to TaskTrackers. Each TaskTracker keeps track of its assigned work units, and reports back results.
- NameNode and DataNode are used for data storage. They are part of HDFS, and not needed when other file systems are used. The NameNode manages metadata and file permissions. It initiates replication when data is imported. Data itself is stored in chunks on nodes; when it needs to be moved it is sent directly between nodes.

Its architecture shows why Hadoop cannot be considered highly available. In fact, there are plenty of nodes that run DataNode and TaskTracker processes. When one of them fails, its work unit is simply reassigned to another node. As data chunks are also stored in other parts of the cluster, data will also remain available. However, the master node with JobTracker and NameNode is not redundant, and holds the single point of failure.

[5] `http://aws.amazon.com/`

[6] `http://aws.amazon.com/elasticmapreduce/`

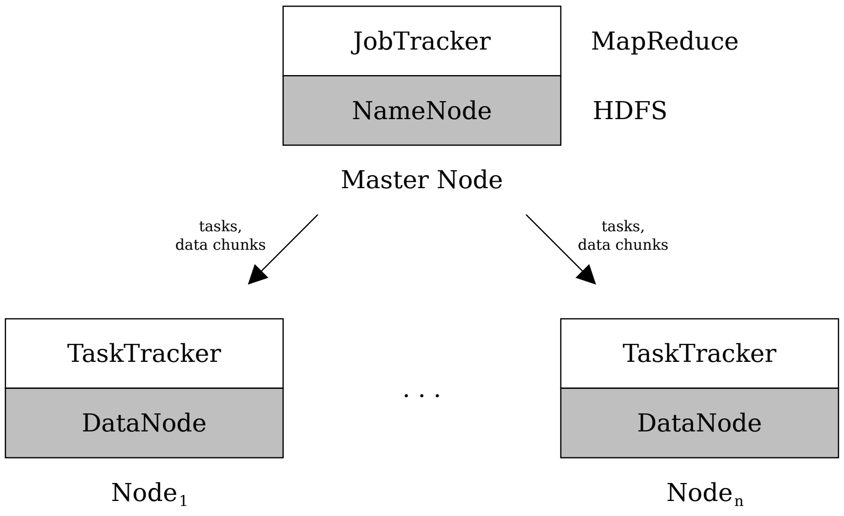


Figure 6.2: Hadoop follows a master/slave architecture.

Projects that Utilise Hadoop

A number of projects have emerged around Hadoop that either build on Hadoop or add new features. Below, we mention three interesting projects in brief.

- *HBase*[7] is a distributed data storage system modelled after BigTable[8]. Its goal is very large tables, with billions of rows and millions of columns.
- *Hive*[9] provides an SQL-like query language for Hadoop. Queries get converted to MapReduce programs, which are then executed on the cluster.
- *Mahout*[10] is a machine learning tool kit for MapReduce, and focuses on document classification and data mining. Distributed implementations of common algorithms are available, for example k-means clustering [1].

For our implementation of a record linkage system we used the Cascading framework, which we detail below.

6.1.3 Cascading

Expressing a data-processing workflow in "raw" MapReduce is challenging: The workflow has to be divided in a series of jobs and their execution order must be synchronised, i.e. a new step in the workflow can only begin after data from previous steps becomes ready. With growing complexity, this quickly gets error-prone. Also,

[7]`http://wiki.apache.org/hadoop/Hbase`
[8]`http://labs.google.com/papers/bigtable.html`
[9]`http://hive.apache.org/`
[10]`http://mahout.apache.org/`

when changed requirements make it necessary to adjust the implementation, many modifications to individual functions as well as how these are connected have to be made.

The Cascading framework[11] provides abstraction from this by hiding the MapReduce layer. Instead of having to think in mappers and reducers, a workflow is modelled in more natural terms. For example, Cascading has *taps*, *pipes*, *sinks* and *tuples* [7].

Data is read from taps to pipes, where it is represented by tuples. A tuple consists of user-defined fields that store values, e.g. artist name or track title. For processing, functions, aggregators and filters are applied to a pipe to shape its tuple stream. Pipes can be combined to bring together results from different operations. Results are finally written to a sink.

A combination of pipes that transforms data from one or more source taps and writes the result to a sink form a *flow*. All flows of a program are called a *cascade*. During later execution the flows are transparently translated to a series of MapReduce jobs [6]. Cascading will take care of synchronisation and schedule the right order of execution. For this, data dependencies are identified. Jobs that do not share dependencies can be run concurrently, while other jobs might have to wait until their input data is ready.

As Cascading makes it very easy to write programs for MapReduce, it has attracted attention at Last.fm. After initial tests, Cascading has been used to rewrite existing services and to implement new services. General consensus is that even complex workflows are easily expressed in Cascading's logical model and program code remains more readable. Besides, no additional software needs to be installed on the Hadoop cluster; programs simply have to include a small library. Existing MapReduce jobs can be reused in a program without modification.

6.2 Notes on our Implementation

Choosing a framework allowed us to rapidly adapt our combinations from Chapter 5 to MapReduce while Cascading handled details, such as job scheduling, managing temporary data and partitioning, sorting or distributing of results between individual MapReduce jobs.

For testing purposes, we installed Hadoop in stand-alone mode on our development machine. We opted for the preconfigured *CDH3* Hadoop distribution from consulting firm Cloudera[12], as it is available for a number of Linux distributions. In stand-alone mode, all map and reduce functions are run inside a single Java Virtual Machine.

[11] http://www.cascading.org/

[12] http://www.cloudera.com/

We were able to directly reuse code from `affiligator` for normalisation and scoring. However, we did not continue using wrapper classes for catalogue and affiliate data (cf. 4.2.1), as we found that Cascading's tuple class made it easy enough to access and change field values. Even though the inverted index demanded a complete rethink, we consider the entry threshold for learning Cascading and adapting our combinations to a workflow as low.

Our prototypical implementation for Hadoop was named `affiligator-cascading`. In the following sections we will present some aspects and thoughts on our implementation.

6.2.1 Normalisation

Data is read as strings to a pipe from a location in HDFS or the local file system and split in fields. There is one pipe for each data set. At the same time, the entry is tested whether all fields for scoring are present. Malformed entries will be logged to standard out and are not considered for further processing. A new field is added that marks where the data comes from, i.e. music catalogue or from an affiliate. This information will be needed for blocking.

Adding the index value to affiliate entries was more difficult than in a single-threaded environment: A MapReduce job is executed on several nodes. Each node runs the program in isolated runtime environments. Keeping track of a running index value is not feasible. As a solution, we calculate the MD5 sum of the processed string, and add it to the tuple to distinguish entries.

After string simplifiers are applied, all tuples in both pipes will feature the same format, with fields for ID, artist name, track title, URL and origin.

6.2.2 Blocking

Adapting the inverted index proved to be the most challenging task. We finally settled for a three-step process for blocking which is shown in detail in figure 6.3 in a simplified form:

1. Each entry is tokenised. For every token, a new tuple will be created that stores token, entry ID, the number of tokens the entry was split in, as well as the entry's origin (6.3a).
2. Both pipes with tokenised catalogue and affiliate tuples are merged into a new pipe which is then grouped by token (6.3b). The tuples in every resulting group are divided in two subsets according to their origin and the Cartesian product of both sets is calculated. Thus, all entry pairs that share at least one token are found.

3. Tuples are sorted and grouped again, this time by catalogue and affiliate ID. We are now able to directly identify how many common tokens an entry pair has by looking at their group size (6.3c).

Tuples are filtered by token distance (see condition 4.6) which leaves entry pairs that are suitable for scoring. However, before we can continue with the next step, artist names, and track titles from the pipe that holds the normalised data must be added first. Cascading offers aggregator functions that mimic well-know types of joins from relational databases (e.g. inner, outer, or full outer join) [5]. However, we tried to avoid joining data on fields for as long as possible. This is because a join, when expressed in MapReduce, will involve a costly reduce phase.

We also experimented with a denormalised view of the data to avoid any need to join data from other pipes. As this led to an explosion of temporary data, we did not put this idea into practice.

6.2.3 Scoring and Matching

Scoring and matching do not involve any other actions than previously described: Values for artist names and track titles are selected from tuples, and used to calculate similarity scores. Matches are afterwards determined by removing all tuples that have a score smaller than the set threshold. Final results are again written to HDFS or the local file system. These will be split over several files with one file per reduce process.

6.2.4 Optimisation for the Dice Similarity Coefficient

In addition to the customisable `affiligator-cascading` project that we have described, which can be used with different similarity metrics, we also created another adaptation (named `affiligator-cascading-dice`) where the workflow was optimised for the Dice similarity coefficient.

In order to determine similarity with this metric, artist names and track titles are not needed. Instead, a score can be computed just with the numbers for common tokens and how many tokens each entry was split in (compare section 4.2.2). We thereby saw the possibility to move joining data to an even later point in the process, namely after entry pairs had been scored and results with a score below the threshold had been removed. As parameters we used combination D from section 5.5: 3-grams for blocking and scoring, token difference $d = 20$, and threshold $t = 0.8$.

6.3 Evaluation and Experiments

For testing both our adaptations we chose two current data sets from Last.fm: An excerpt from the music catalogue with about 6.3 million entries, and a delivery from an affiliate with about 11.4 million entries.

6.3.1 Test Setup

We conducted our experiments on Last.fm's Hadoop cluster. Currently, the cluster consists of 44 nodes. Each node has two Intel Xeon L5520 quad-core CPUs running at 2.27GHz, 16GB of RAM and 8TB storage. Hadoop is provided by Cloudera's CDH3 distribution with some additionally applied patches, mostly for performance improvements.

6.3.2 Results

We made sure that our projects functioned correctly with the `popcat` and `randaff` data sets that we had used in our previous evaluation in Chapter 5. Then, we proceeded with the large data sets.

During our adaption we had thought that joining the normalised data was the most complex operation in the workflow. However, our small-scale tests already showed that the blocking step took much longer than we had expected and proved to be a major performance bottleneck.

This became even more clear when we directly compared results for combination B (uses the Dice similarity coefficient) with `affiligator-cascading-dice`: We had considered the latter to be "optimised", however, it needed noticeably more time to match data sets. We attributed this to the increased n-gram volume that must be sorted and grouped, due to a smaller $n = 3$.

Although the existing systems for ingesting playlinks and buylinks, which are employed in production use at Last.fm, cannot be compared to our prototypical adaptations, we used their mean running time of about seven hours as an upper bound for tests with large data sets. Unfortunately, we were not able to complete just one matching in this period and had to cancel our tests after six hours when execution was still busy with blocking. We also saw very large amounts of temporary data being created.

For an impression of the successful execution of `affiligator-cascading` on the Hadoop cluster, we conducted several matching tests with decreased data set sizes (the largest sets had 525,541, and 259,413 entries). Running times for all tests are plotted in figure 6.4 (we were only able to make two measurements for combination D). It clearly shows

Combination	Similarity metric	Overview			Sampled matches	
		$\sum$	Exact	Approx.	Correct	False positives
A	Levenshtein	6467	4775	1692	83	17
B	Dice	10,348	4775	5573	46	54
C	Cosine	12,166	4775	7391	51	49

Table 6.1: Quality of matching results for our MapReduce implementations. Used data sets had 525,541 and 259,413 entries.

that running time is affected most by the set gram length for blocking, and less by how many similarity comparisons have to be made.

We also analysed the result quality. For this, matches were divided in exact and approximate matches, and 100 approximate matches were sampled and manually classified in correct matches and false positives (see table 6.1).

6.4 Summary and Discussion

MapReduce is an interesting concept for data-centric applications as it makes it very convenient to distribute extensive computations over a cluster of inexpensive computers. By using the Cascading framework we were able to rapidly adapt our `affiligator` system to Hadoop, without having to program "raw" map and reduce functions.

Normalisation, tokenisation, and scoring of entries are small and independent tasks that can efficiently be expressed in the MapReduce model.

Our solution to the important blocking step, however, presents a major performance bottleneck. In particular, we are especially unsatisfied with two things: The large body of tokens must be sorted and grouped first. Then, after candidate pairs were filtered by common tokens, the normalised entries have to be joined. This takes very long and creates an unacceptable amount of temporary data, even for small data sets.

6.4.1 Possible Improvements

We have identified possible improvements that solve the painful blocking step more efficiently. In the absence of time, however, these were not implemented, nor further evaluated.

Reducing the Token Body

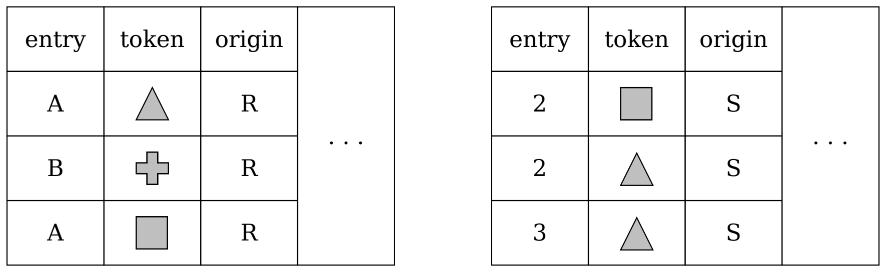
The *PPJoin* and *PPJoin+* algorithms [35] are supposed to yield good results for detecting duplicates in data sets. Both reduce size and cost for identifying entry pairs for scoring by only indexing some tokens of an entry (those that are part of the entries' variable-length *prefix*). Candidate pairs are filtered further by introducing constraints, for example, for positional distance of tokens. Application of both algorithms to MapReduce has been demonstrated [32, 31].

External Systems for Identifying Candidates

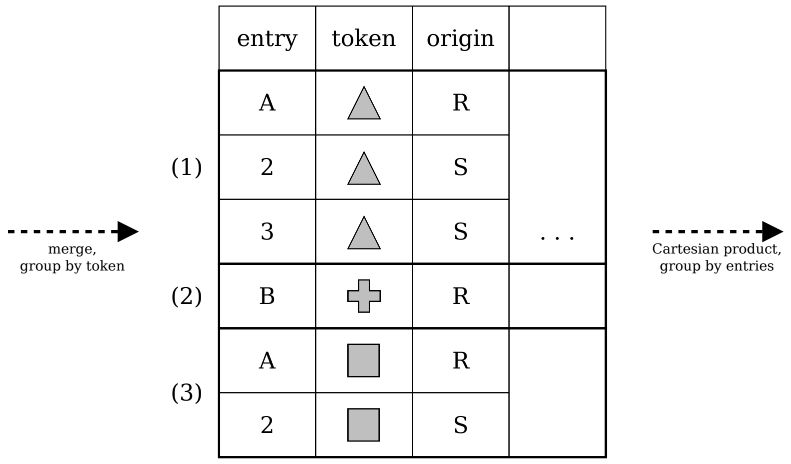
Rather than relying on MapReduce for every step of the record linkage process, it should only be used for tasks that do not share dependencies with other data. For this reason, we propose the following sketch: The index structure is kept on an external system that can either identify and prepare candidate pairs or that allows for fast querying of candidates.

There is a broad choice of existing "building blocks" that could be used, for example relational databases [13] or a (distributed) key-value storage system like *Memcached*[13] [37]. This would allow map functions to exchange intermediary results or to query for specific data. Moreover, it is not feasible to normalise and tokenise the whole local data collection for every matching. With an external system, preparing and indexing of local data could be done in a separate process, for example once per week.

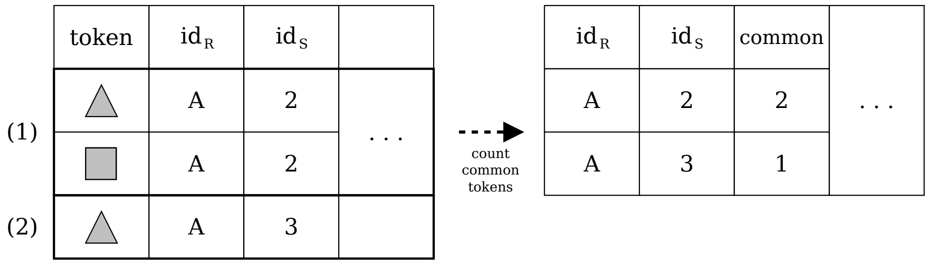
[13]`http://www.memcached.org`



(a) Tokenised data sets.



(b) Entries are merged into one pipe.



(c) Common tokens are counted.

Figure 6.3: The employed three-step process for blocking tokenised data sets R and S. Data sets are tokenised and merged. The origin of an entry is stored in a field. Tokens are grouped and entry pairs that share a common token are identified. Finally, entries are sorted again to find out how many common tokens an entry pair shares.

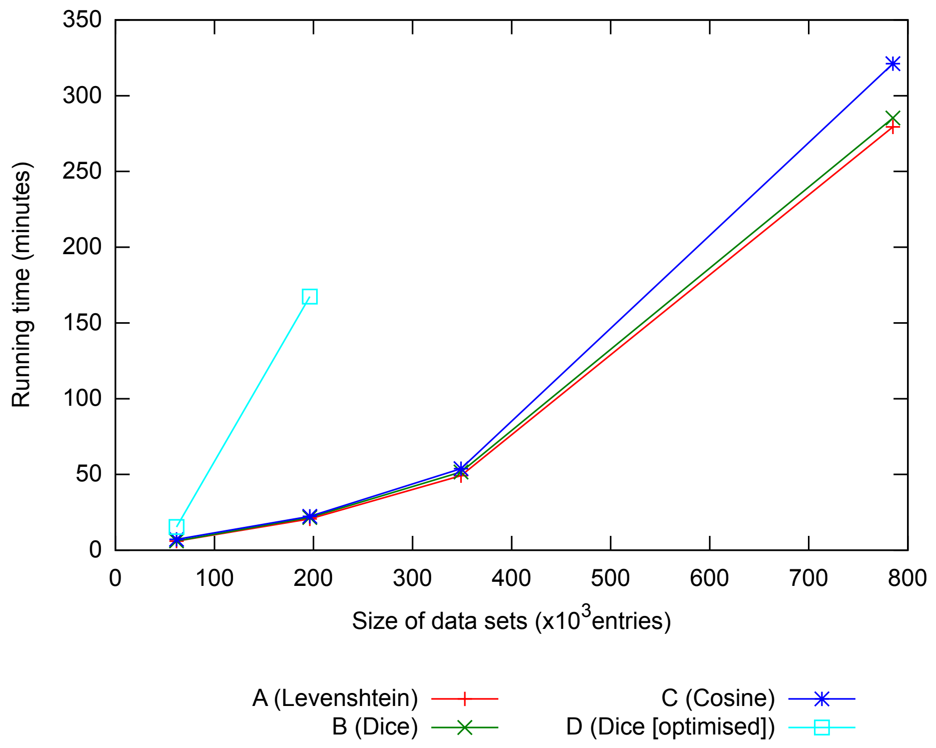


Figure 6.4: Running times for tested combinations and different data set sizes.

7 Conclusions and Future Work

This dissertation has described the work necessary to produce a solution for matching entries in large data sets with MapReduce.

Our initial aims were (1) to give an overview of the record linkage process, (2) identify optimal combinations of techniques and (3) adapt these combinations to MapReduce (cf. section 2.3). It has shown over the course of our work, however, that we could not fully meet all of our requirements. In this chapter we summarise our progress and conclude with a discussion of future work.

7.1 Summary

We used Chapters 2 and 3 for a thorough introduction to record linkage and its individual steps (normalisation, blocking, scoring and matching), and how growing scale adds another level of complexity to the task of identifying entries that represent the same logical entity in data sets.

This formed the basis for coming up with a selection of techniques for each step of the record linkage process in Chapter 4. We were especially interested in a spatial index approach, where string data is mapped to a vector space; entries that share a certain similarity can then be found by comparing how far their vectors lie apart. We also derived our own data sets from real-world data that is commonly used at music recommendation service Last.fm

Our following evaluation in Chapter 5 showed that it was cheaper to build an inverted index directly from string data than to do a costly vector-mapping. At the same time, by evaluating each step of the record linkage process separately we were able to identify several combinations of techniques that deliver best results for the used type of data: We found one combination to be better for accurate results, while the remaining combinations favoured a higher recall over precision.

We proceeded to adopt our findings to the MapReduce paradigm and Hadoop in Chapter 6. MapReduce is popular for data-centric tasks. Nevertheless, it demands that a program operates within certain restrictions. Therefore, we first explained the concepts of MapReduce and the architecture of Hadoop in greater detail. We used the Cascading framework to implement our solution, and its logical model of data flows allowed us to rapidly develop a prototypical implementation, which we then tested with large data sets. Unfortunately, the performance was poor and other than we had hoped. This was due to the way we adapted the blocking step, for which we immediately presented possible improvements.

7.2 Future Work

As mentioned in section 6.4, we see in MapReduce an excellent tool for processing large data sets, as it allows to conveniently distribute computation over a cluster of machines. For this, however, a problem must be dividable in small, independent tasks. In terms of record linkage, this holds true for normalisation, tokenisation, and scoring of entries.

Our main concern is the slow and painful blocking step. Hence, future work should focus on identifying better solutions than the one we employed. Apart from improvements that were already detailed in 6.4.1, we propose investigating the following fields.

7.2.1 Vector Spaces and Spatial Data Structures

We are interested in researching other methods that can either act as replacements for components in the approach we used as a spatial index (4.2.3) or that point in different directions.

Mapping Entries to Vectors

For mapping string to vectors, we used StringMap, which is an interesting approach as it maps strings to a true Euclidean space of arbitrary dimensionality m. Mapping a large amount of strings, however, was time-consuming.

Another method worth researching is *random indexing* [27]. It generates sparsely populated vectors of very high dimensionality and surprises by its sheer simplicity: Strings are again decomposed in tokens and each unique token is assigned a random *index vector*. To map a string to a vector, all its tokens' index vectors are simply added. Distances between the resulting vectors will (approximately) reflect string similarity although index vectors were chosen at random.

Grouping Vectors

Strings that share a certain similarity will have vectors that lie close together. For blocking similar strings, we have classified their vectors by using a clustering algorithm. Other than that, spatial data structures were designed for storing m-dimensional data (e.g. R-Trees) and can be queried for entries that are within a given distance to an object.

It has been shown that spatial problems can be applied to MapReduce, for example k-means clustering [1], but also constructing R-Trees [3]. On the other hand, it should be evaluated first if satisfactory results can be achieved by creating the index on a single machine, where it also could be kept in memory for better performance.

7.2.2 Similarity-preserving Hashing

Instead of creating an inverted index, each entry from a data set can be hashed. The hash function that is employed, however, is not exact, but approximate. This means that the same key can be computed for two entries, even though they are not equal. Similar entries are more likely to have the same key than dissimilar entries.

Candidates for scoring could easily be identified with MapReduce by grouping entries with the same key. All entries would still have to be sorted and send to reducers but in this case the number of entries is smaller than when tokenised. Also, a separate step for joining normalised data is not needed.

A number of approximate hashing techniques exist, for example *Minhash*, and *LSH* (locality-sensitive hashing) [24]. We briefly experimented with a simple approximate hashing method in preparation to this dissertation, but results were not conclusive, as the hashed artist name and track title combinations are rather short. It would have to be evaluated if there are similarity-preserving hashing techniques that return good results for short strings and up to what edit distance the same keys are returned.

Bibliography

[1] Apache Mahout Wiki: *K-Means Clustering.* – URL https://cwiki.apache.org/confluence/display/MAHOUT/K-Means+Clustering. – Zugriffsdatum: 29/09/2011

[2] Bilenko, Mikhail ; Mooney, Raymond ; Cohen, William ; Ravikumar, Pradeep ; Fienberg, Stephen: Adaptive name matching in information integration. In: *IEEE Intelligent Systems* 18 (2003), Nr. 5

[3] Cary, Ariel ; Sun, Zhengguo ; Hristidis, Vagelis ; Rishe, Naphtali: Experiences on processing spatial data with mapreduce. In: *Scientific and Statistical Database Management*, 2009

[4] Cohen, William W. ; Ravikumar, Pradeep ; Fienberg, Stephen E.: A Comparison of String Distance Metrics for Name-Matching Tasks. In: *Learning* 20 (2003), Nr. 1

[5] Concurrent, Inc.: *Joining a Tuple Stream.* – URL http://www.cascading.org/userguide/html/ch03s02.html#N20308. – Zugriffsdatum: 01/08/2011

[6] Concurrent, Inc.: *MapReduce Job Planner.* – URL http://www.cascading.org/1.2/userguide/html/ch10.html. – Zugriffsdatum: 01/08/2011

[7] Concurrent, Inc.: *User Guide.* – URL http://www.cascading.org/userguide/html/. – Zugriffsdatum: 01/08/2011

[8] Davis, Mark ; Whistler, Ken: *Unicode Normalization Forms.* – URL http://www.unicode.org/reports/tr15/. – Zugriffsdatum: 18/05/2011

[9] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *Communications of the ACM* 51 (2004), Nr. 1

[10] Elmagarmid, Ahmed K. ; Ipeirotis, Panagiotis G. ; Verykios, Vassilios S.: Duplicate Record Detection: A Survey. In: *IEEE Transactions on Knowledge and Data Engineering* 19 (2007), Nr. 1

[11] Fellegi, Ivan P. ; Sunter, Alan B.: A Theory for Record Linkage. In: *Journal of the American Statistical Association* 64 (1969), Nr. 328

[12] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google File System. In: *ACM SIGOPS Operating Systems Review*, 2003

[13] Gravano, Luis ; Ipeirotis, Panagiotis G. ; Pietarinen, Lauri ; Koudas, Nick ; Muthukrishnan, Shanmugauelayut ; Pietarinen, Lauri ; Srivastava, Divesh: Using q-grams in a DBMS for Approximate String Processing. In: *Computer* 24 (2001), Nr. 4

[14] Hadoop Documentation: *HDFS Architecture Guide.* – URL `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. – Zugriffsdatum: 09/08/2011

[15] Hadoop Documentation: *HDFS Users Guide.* – URL `http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html`. – Zugriffsdatum: 09/08/2011

[16] Hadoop Wiki: *Frequently Asked Questions.* – URL `http://wiki.apache.org/hadoop/FAQ`. – Zugriffsdatum: 09/08/2011

[17] Hadoop Wiki: *How Map and Reduce operations are actually carried out.* – URL `http://wiki.apache.org/hadoop/HadoopMapReduce`. – Zugriffsdatum: 09/08/2011

[18] Hadoop Wiki: *Project Description.* – URL `http://wiki.apache.org/hadoop/ProjectDescription`. – Zugriffsdatum: 09/08/2011

[19] Hernández, Mauricio A. ; Stolfo, Salvatore J.: The merge/purge problem for large databases. In: *ACM SIGMOD Record* 24 (1995), Nr. 2

[20] Ji, Shengyue ; Li, Guoliang ; Li, Chen ; Feng, Jianhua: Efficient interactive fuzzy keyword search. In: *Proceedings of the 18th international conference on World wide web WWW 09* (2009)

[21] Kolb, L. ; Thor, A. ; Rahm, E.: Multi-pass sorted neighborhood blocking with MapReduce. In: *Computer Science-Research and Development* (2011)

[22] Li, Chen ; Jin, Liang ; Mehrotra, Sharad: *Supporting efficient record linkage for large data sets using mapping techniques.* 2006. – URL `http://www.ics.uci.edu/~chenli/pub/2006-ljm.pdf`. – Zugriffsdatum: 07/06/2011

[23] MacKay, David J C.: *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003. – 285–289 S

[24] Moulton, Ryan: *Simple Simhashing: Clustering in linear time.* – URL `http://knol.google.com/k/ryan-moulton/simple-simhashing/3kbzhsxyg4467/6`. – Zugriffsdatum: 29/09/2011

[25] Navarro, Gonzalo: A guided tour to approximate string matching. In: *ACM Computing Surveys* 33 (2001), Nr. 1

[26] Needleman, Saul B. ; Wunsch, Christian D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. In: *Journal of Molecular Biology* 48 (1970), Nr. 3

[27] Sahlgren, Magnus: An introduction to random indexing. In: *Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005*, 2005

[28] Salton, G ; Wong, A ; Yang, C S.: A vector space model for automatic indexing. In: *Communications of the ACM* 18 (1975), Nr. 11

[29] Statistics New Zealand: *Data Integration Manual.* Crown Copyright Statistics New Zeland, 2006

[30] Tveit, Amund: *Mapreduce & Hadoop Algorithms in Academic Papers (4th update – May 2011).* – URL `http://atbrox.com/2011/05/16/mapreduce-hadoop-algorithms-in-academic-papers-4th-update-may-2011/`. – Zugriffsdatum: 29/09/2011

[31] Vernica, Rares ; Carey, Michael J. ; Li, Chen: Efficient parallel set-similarity joins using MapReduce. In: *Proceedings of the 2010 international conference on Management of data SIGMOD 10* (2010)

[32] Wang, C ; Wang, J ; Lin, Xuemin ; Wang, W ; Wang, Haixun ; Li, H ; Tian, W ; Xu, J ; Li, R: MapDupReducer: Detecting Near duplicates over Massive Datasets. In: *Proceedings of the 2010 international conference on Management of data*, 2010

[33] Winkler, William E.: Overview of Record Linkage and Current Research Directions / U.S. Bureau of the Census. 2006. – Forschungsbericht

[34] Winkler, William E. ; Thibaudeau, Yves: An application of the Fellegi-Sunter model of record linkage to the 1990 US decennial census / U.S. Bureau of the Census. 1990. – Forschungsbericht

[35] Xiao, Chuan ; Wang, Wei ; Lin, Xuemin ; Yu, Jeffrey X.: Efficient Similarity Joins for Near Duplicate Detection. In: *Proceeding of the 17th international conference on World Wide Web WWW 08* 36 (2008), Nr. 3

[36] Yang, Zhenglu ; Yu, Jianjun ; Kitsuregawa, Masaru: Fast algorithms for top-k approximate string matching. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010

[37] Zhang, Shubin ; Han, Jizhong ; Liu, Zhiyong ; Wang, Kai ; Feng, Shengzhong: Accelerating MapReduce with Distributed Memory Cache. In: *2009 15th International Conference on Parallel and Distributed Systems* (2009)