Learning From the Past of a Digital Library

Using Historical Metadata to Study the Development of Collections

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
am Fachbereich IV der Universität Trier

vorgelegt von **Dipl.-Inf. Florian Reitz**

im Juli 2017

Betreuer: Prof. Dr. Bernd Walter, Universität Trier

Danksagung

Beim Schreiben dieser Danksagung ist mir noch einmal bewusst geworden, wie viele Menschen mich beim Abfassen dieser Arbeit unterstützt haben und ohne die dieses Werk nicht existieren würde. Ich bin allen zutiefst dankbar – denen, die weiter unten gelistet sind, und denen, die ich hier nicht nennen kann.

Mein besonderer Dank gilt Prof. Dr. Bernd Walter, der diese Arbeit betreut hat und immer mit Rat zur Seite stand. Darüber hinaus danke ich meinen aktuellen und ehemaligen Kollegen an der Universität Trier und dem Leibniz-Zentrum für Informatik in Wadern für zahlreiche Diskussionen, Anregungen und Hilfestellungen. Besonders nennen möchte ich hier Marcel Ackermann, Sebastian Baltes, Fabian Beck, Peter Birke, Michael Ley, Oliver Hoffmann, Peter Weißgerber und Ralf Schenkel der sich freundlicherweise als Zweitgutachter zur Verfügung gestellt hat.

In diese Arbeit sind eingeflossen die Ideen und Beiträge von studentischen Mitarbeitern und Absolventen. Besonders möchte ich mich hier bei Thomas Kirsch und Willi Hornig bedanken. Ich danke Stefanie von Keutz für das detaillierte Korrekturlesen dieser Arbeit. Ohne ihre Hilfe wäre es wohl nicht möglich gewesen diese Arbeit auf Englisch zu verfassen. Nicht vergessen möchte ich Agnes Jacoby, die mich stets in administrativen Fragen unterstützt hat.

Abschließend möchte ich meiner Familie, insbesondere meinen Eltern Ewald und Mathilde, sowie meinen Freunden und Bekannten danken, ohne deren Zuspruch und Motivation diese Arbeit schlicht nicht existieren würde.

Florian Deitz

Trier, den 22. Juni 2017

Zusammenfassung

Digitale Bibliotheken haben sich zu einem zentralen Bestandteil unseres Lebens entwickelt. Sie ermöglichen uns den unmittelbaren Zugriff auf eine vorher unvorstellbare Menge an Informationen. Dank der Unterstützung durch Rechner und der Möglichkeit, Daten (halb)automatisch zu aggregieren, können digitale Bibliotheken beachtliche Größen erreichen, auch wenn sie nur von vergleichsweise kleinen Organisationen bereitgestellt werden. Ein zentraler Aspekt digitaler Bibliotheken sind die Metadaten: Informationen, die die gespeicherten Dokumente beschreiben. Metadaten sind digital verfügbar und können deshalb automatisch ausgewertet werden. In den vergangenen Jahren haben sich viele Studien mit verschiedenen Aspekten der Metadaten beschäftigt. Von zentralem Interesse ist dabei das Auffinden von Fehlern in den Daten, insbesondere im Zusammenhang mit Autorennamen. Diese Studien konzentrieren sich üblicherweise auf die aktuellen Metadaten einer digitalen Bibliothek. Z.B. wird nach Fehlern gesucht, die eine Bibliothek an einem bestimmten Tag X enthielt. In vielen Szenarien ist diese Einschränkung sinnvoll. Soll aber die Frage beantwortet werden, wie die Fehler entstanden sind, die an Tag X gefunden wurden, so muss die Geschichte der Metadaten untersucht werden.

In dieser Arbeit beschäftigen wir uns mit der Frage, wie die Geschichte der Metadaten in einer digitalen Bibliothek untersucht werden kann. Hierzu untersuchen wir zunächst, wie digitale Bibliotheken Informationen speichern und aufbewahren. Anhand dieser Information entwickeln wir eine Systematik zum Beschreiben der vorhandenen historischen Informationen, also Informationen darüber, wie sich Metadaten im Laufe der Zeit verändert haben. Basierend auf dieser Systematik stellen wir ein System vor, das Änderungen an Metadaten identifiziert und semantisch zusammenhängende Änderungen in Blöcken zusammenfasst. Es stellte sich heraus, dass historische Metadaten nur für wenige Bibliotheken vorhanden sind. Dennoch ist es uns gelungen, das System zum Erkennen von Änderungen auf eine Reihe großer Kollektionen anzuwenden.

Ein zentraler Aspekt dieser Arbeit ist das Identifizieren und Analysieren von Änderungen, die Fehler korrigierten. Diese Änderungen beschreiben die vergangenen Versuche, die Datenqualität einer digitalen Bibliothek zu steigern. Wir stellen ein System vor, das bestimmte Typen von Fehlern automatisch aus der Gesamt-

menge aller Änderungen extrahiert und klassifiziert. Basierend auf Korrekturen, die aus dem DBLP-Datensatz extrahiert wurden, erstellen wir Testkollektionen, die insgesamt mehr als 100.000 Fehler-Konstellationen enthalten. Diese Testkollektionen können zum Testen automatischer Fehlersuchverfahren genutzt werden. Darüber hinaus können Korrekturen genutzt werden, um die Eigenschaften von Fehlern zu verstehen. Besonders untersuchen wir hier Fehler im Zusammenhang mit Autorennamen. Wir zeigen, dass viele Fehler in Situationen entstehen, in denen wenig Kontext-Daten zur Verfügung stehen, was erhebliche Auswirkungen auf automatische Fehlerdetektoren haben kann. Wir zeigen außerdem, dass sich Fehler, die aus verschiedenen Kollektionen extrahiert wurden, stark unterscheiden können. In einem kurzen Ausblick untersuchen wir, wie korrigierte Fehler genutzt werden können, um unentdeckte oder zukünftige Fehler zu identifizieren.

Neben der Analyse von Fehlern eignen sich historische Metadaten, um die Entwicklung einer digitalen Bibliothek genauer zu untersuchen. In dieser Arbeit zeigen wir anhand exemplarischer Studien, welche Informationen aus historischen Metadaten abgeleitet werden können. Zunächst untersuchen wir die Entwicklung der DBLP-Kollektion über einen Zeitraum von 15 Jahren. Wir konzentrieren uns dabei auf Änderungen der thematischen Abdeckung verschiedener Felder der Informatik. Unter anderem zeigen wir, wie DBLP sich von einem speziellen Projekt zu einer allgemeinen Sammlung entwickelt hat. In einer weiteren Studie untersuchen wir, wie E-Mails von DBLP-Benutzern sich auf das Finden von Fehlern in DBLP auswirken. Wir zeigen, dass viele Fehlerkorrekturen durch Benutzer angeregt werden. Darüber hinaus können wir Rückschlüsse darauf ziehen, was Benutzer dazu bringt, einen Fehler im DBLP-Datenbestand zu melden.

Abstract

Digital libraries have become a central aspect of our live. They provide us with an immediate access to an amount of data which has been unthinkable in the past. Support of computers and the ability to aggregate data from different libraries enables small projects to maintain large digital collections on various topics. A central aspect of digital libraries is the metadata – the information that describes the objects in the collection. Metadata are digital and can be processed and studied automatically. In recent years, several studies considered different aspects of metadata. Many studies focus on finding defects in the data. Specifically, locating errors related to the handling of personal names has drawn attention. In most cases the studies concentrate on the most recent metadata of a collection. For example, they look for errors in the collection at day X. This is a reasonable approach for many applications. However, to answer questions such as when the errors were added to the collection we need to consider the history of the metadata itself.

In this work, we study how the history of metadata can be used to improve the understanding of a digital library. To this goal, we consider how digital libraries handle and store their metadata. Based in this information we develop a taxonomy to describe available historical data which means data on how the metadata records changed over time. We develop a system that identifies changes to metadata over time and groups them in semantically related blocks. We found that historical meta data is often unavailable. However, we were able to apply our system on a set of large real-world collections.

A central part of this work is the identification and analysis of changes to metadata which corrected a defect in the collection. These corrections are the accumulated effort to ensure data quality of a digital library. In this work, we present a system that automatically extracts corrections of defects from the set of all modifications. We present test collections containing more than 100,000 test cases which we created by extracting defects and their corrections from DBLP. This collections can be used to evaluate automatic approaches for error detection. Furthermore, we use these collections to study properties of defects. We will concentrate on defects related to the person name problem. We show that many defects occur in situations where very little context information is available. This has major implications for automatic

defect detection. We also show that properties of defects depend on the digital library in which they occur. We also discuss briefly how corrected defects can be used to detect hidden or future defects.

Besides the study of defects, we show that historical metadata can be used to study the development of a digital library over time. In this work, we present different studies as example how historical metadata can be used. At first we describe the development of the DBLP collection over a period of 15 years. Specifically, we study how the coverage of different computer science sub fields changed over time. We show that DBLP evolved from a specialized project to a collection that encompasses most parts of computer science. In another study we analyze the impact of user emails to defect corrections in DBLP. We show that these emails trigger a significant amount of error corrections. Based on these data we can draw conclusions on why users report a defective entry in DBLP.

Contents

List of Figures

List of Tables

Chapter 1

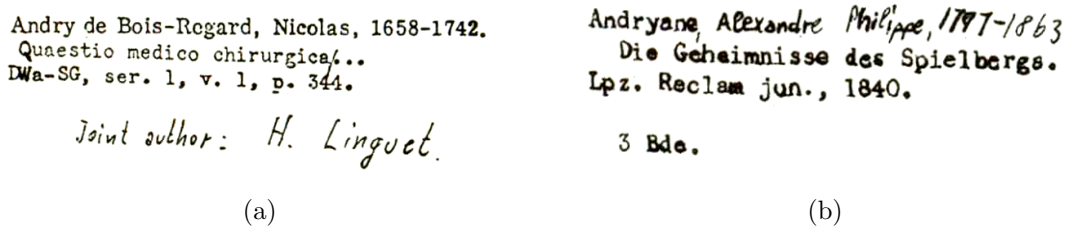
Introduction

Contents

To find a document in a medieval library the user had to know under which topic it was registered. The user would then locate the shelf assigned to that topic and look through all the material that was stored there. When libraries became larger, this simple approach failed. During the 14th century, librarians began to separate documents and retrieval systems [Joc99, 82 pp.]. From each document they created a small record that contained descriptive information such as title and author name. These **metadata** records were placed in catalogs. Each catalog was sorted in a specific way, for example, by title. The user could now use the catalogs to determine the exact position of a document. Since then, the metadata records and the catalogs which are built on top of them ensure that documents can be located in a library. Creating and organizing bibliographic metadata became one of the central tasks of a library. As the *National Information Standards Organization* points out:

> "Metadata is key to ensuring that resources [the documents] will survive and continue to be accessible into the future." [NIS04, p. 1]

To fulfill this goal, metadata records must have a high quality. Libraries use complex rulebooks and extensive staff training to ensure that new records are created with as few errors as possible. However, it is often necessary to modify existing records. Figure 1.1 shows two records from the National Union Catalog (NUC) pre 1956 imprints [Ame72]. We can see different modifications. For example, in (a) the title was changed from *Ouaestio* to *Quaestio*. It is obvious that this correction was



(a) (b)

Figure 1.1: Modified index cards from the National Union Catalog.

Source: NUC [Ame72, Volume 16 page 532]

necessary: in a catalog sorted by title it would have been very unlikely to find this book as it would have been listed under O instead of Q.

If we look through the NUC imprints, we can see that modifications occur frequently. This is not surprising given the large amount of metadata and the high potential for defects. We can safely assume that all metadata collections of active libraries contain similar modifications to the data. In this work, we will argue that studying the modifications can provide insight into different aspects of a library's data quality. Consider again the modifications shown in Figure 1.1. We might ask the following questions:

Were the defects isolated or do they occur frequently? If the confusion of O and Q was an isolated mistake, a simple correction is sufficient. If this particular error occurs frequently, specific staff training or even a reorganization of the record creation process might be necessary.

When were the modifications carried out? The modifications in the example do not have a timestamp. It is unclear how long the defects persisted. Example (a) has multiple corrections. Were they corrected together or at different times? Perhaps the modification O to Q is a side product of a curator adding the new author.

What was the nature of the defect? While the O,Q error in (a) is easy to understand, the nature of the modifications in (b) are less clear. This is because the original text was erased and the new text written on top of it. We do not even know if this is a modification or if a staff member replaced a section that was difficult to read.

What caused the defect? The reason for the initially missing author in (a) could be that this record was imported from another library that frequently forgets authors. If many modifications of the same type are done to related records, an investigation might become necessary.

Are there other modifications? The NUC was created by copying index cards. Instead of annotating an index card it is also possible to create a new one

which shows no sign of modification. It is possible that (a) or (b) have previous modifications that are hidden this way.

Was the modification carried out completely? In (a), an author is added. In the NUC this means that a separate entry for this person had to be added to another part of the catalog. From the information here, we cannot see if this was done. Even if we find an entry for *H. Linguet* now, we do not know if this entry was added at the time of this modification or later. In (b), year of birth and year of death were added to the author name. It is very likely that this was done to separate this person from another author with the same name. For this modification to be consistent, all publications of this author had to be modified in the same way.

To answer these questions a structured analysis of modifications to metadata records is needed. This analysis must be carried out on a reliable data source that provides sufficient historical data. While searching for and removing of errors in metadata records is a very active field of research, past modifications have received little attention. This is regrettable as the number of modifications is likely to increase. The central reason is the advent of truly **digital libraries**. While digitized metadata records exist since the 1950s, the full potential has only been used recently. The amount of data exchange increases and libraries can now compare their records to the records created by others. More and more projects encourage their users to report defective data and provide the means to do so. Algorithmic solutions to detect defects, particularly in the field of named entity recognition, are now commonplace.

In this work, we attempt to use modifications to gain insights into libraries. We will concentrate on digital libraries as they more readily provide the data that we need. However, with respect to metadata, the differences between classical and digital libraries become less and less visible. Most of the results of this work can be transferred to metadata of classical libraries, provided that sufficient data on the history of records is available.

1.1 Temporal Aspects of Metadata

In this work, we study modifications to bibliographic metadata in the course of time. To this goal, we examine the same record at different dates and analyze how it has changed. Consider this small example of a record for the play *Othello* by William Shakespeare. Assume that we observed this record at two different days and obtained the following data:

```
title: Othello
creator: William Shakespeare
coverage: 16th century
first_publish_date: 1621
place: Venecia
pages:  120
```

January 1, 2014

```
title: Othello
creator: William Shakespeare
first_publish_date: 1622
coverage: 16th century
place: Venecia (Italy)
subject:  English literature
subject:  drama
```

January 10, 2014

Obviously, the two versions differ significantly. More precisely:

- Two *subject*s are added. (green text)
- *pages* is deleted. (red text)
- The values of *first_publish_date* and *place* are modified. (orange text)
- The order of some data elements has changed.

Modifications like these are done for different reasons. We assume that the *intent* to add the two subjects was to provide more information to facilitate searching. Deleting the page information does not support searching. The intent here might have been to remove defective data. The modification of *first_published_date* appears to be a correction of a wrong value while the modification of *place* is a specialization. An intent to modify metadata can affect several records. Assume that a data curator wants to replace all occurrences of *Venecia* with more precise information. In this case, she would have to modify all records containing this subject. In this work we attempt to group modifications by reconstructing the intent of the data curator.

Modifications, for whatever reason they are done, define a timeline that describes the history of a record. Assume that our example record was created on January 1, 2014 and modified on January 10, 2014. Then this record is related to those dates. The relation is specific for a collection. If the same record is created in another library, we would obtain a different timeline. We call these dates **internal time**. It has to be separated from temporal information that is provided by the document itself, what we call **external time**. The fact that the work was created in 1622 does not affect the handling of the metadata record in the library. It is document specific, e.g., if handled properly all libraries should list the same publication date for this document. Studies on external time of documents in a digital library are frequent. Fenlon and Varvel [FJ13], for example, discuss the importance to evaluate how good digital libraries cover primary source material (pictures, newspaper articles ...) from different times and regions. This is related to external time as it does not matter when a document was added to the collection, it is only important at what time the document was created. The evaluation proposed by Fenlon and Varvel might be extended in a way that it considers the changes to the coverage over time.

This provides insight into whether and how the coverage has improved. We will see in this work that one reason for missing studies on internal time is the difficulty to obtain this information.

1.2 Objectives and Contributions

While many libraries readily share the current versions of their records it is difficult to obtain historical versions. If historical data is available it is often incomplete or there are other issues that hinder or inhibit certain studies. There is also no commonly accepted exchange standard so the records are provided in very different forms. Any framework for modification detection must not depend on a specific metadata definition such as AACR2 or RDA but must be able to integrate very different data formats.

> ***Objective 1:***
> Determine if internal temporal data can be found for a sufficient number of projects and categorize the available data sources. Specify a general data format and import the available data. Specify a framework to describe the properties of the historical data.

We will show that temporal data can be found for some projects. However, we will see that extensive preprocessing might be necessary and that the temporal properties of the data sets differ in many relevant aspects. In this work we will

- define a taxonomy for the classification of collections based on their temporal attributes.
- present an observation-based framework for the processing of temporal data. The framework takes temporal properties into account and can handle data from very different sources.

If data is available, modifications must be located and categorized. Given the size of many collections this has to be done automatically. The challenge is that there are very different types of modifications and that they can appear at the same time. As pointed out in the initial example, modifications can affect several records at once. We want to detect those semantic units as well.

> ***Objective 2:***
> Create a categorization model for modifications and develop a framework to detect them automatically. Group semantically related modifications together.

The model we will present consists of four steps which group local modifications into increasingly larger semantically related units. We will

- discuss the different steps of the model and how they can be extracted automatically from the underlying data.
- present an implementation of the model which can handle collections with up to 48 million records.

In this work, we will concentrate on the development of metadata. The history of the documents themselves is outside the scope of this work.

Some modifications are part of a natural development of metadata. For example, many libraries change their signature system after a while. The new signatures must be added to the records. However, some modifications remove errors from the records. These modifications are important. For example, knowledge on corrected defects can also be used to determine the success of quality management in a digital library. Known defects can be used to create test collections that reflect the data situation in a specific library.

> ***Objective 3:*** Automatically identify modifications that corrected errors. Make use of previous corrections to understand typical data defects. Explore how this knowledge can be used to improve data quality.

We will show that some types of defect corrections can be extracted automatically. Based on these corrections, we will create test collections which can be used to evaluate defect detection and correction algorithms. We will analyze properties of the test collection which are central to most defect detection algorithms. For the correction we will show that:

- properties of defects depend on the underlying digital library and that defect detectors might need adjustment to this.
- the majority of corrections appear to be difficult to handle for algorithmic approaches. In particular, we show that many defects occur in situations where there is little information available.
- past corrections can – within limits – be used to predict future defects in a digital library.

We make the test collections freely available so that they can be used to evaluate author disambiguation algorithms.

Besides defect correction, historical metadata can provide insights into the development of a digital collection. In the simplest form, this could be the number of documents in the collection at a specific point of time.

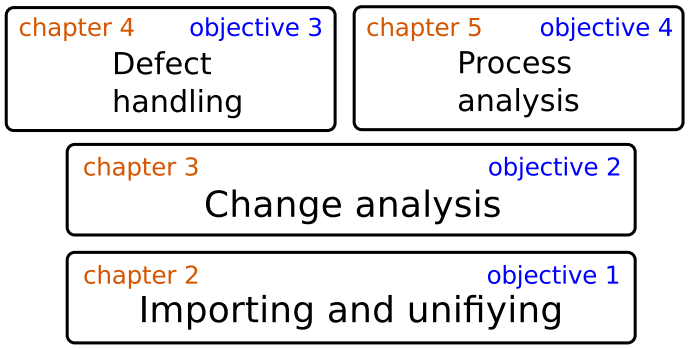


Figure 1.2: General structure of this thesis.

> ***Objective 4:*** Examine how information on modified metadata can be used to describe the history of a collection.

We will conduct two studies that use temporal data to better understand two different aspects of the DBLP collection:

- We will show how the thematic coverage of DBLP has changed over time. This gives us insight into the history of the collection but might also be relevant to judge work based on DBLP.
- We will show that emails from users contribute significantly to detecting defective data in DBLP.

The objectives will be addressed in different chapters of this thesis.

1.3 Outline

The structure of this work follows the objectives outlined above. Each chapter addresses a specific objective. Figure 1.2 gives an overview on the general structure of this thesis.

In Chapter, 2 we discuss possible sources of temporal data and how to categorize them. We introduce an observation-based model for dynamic metadata and present different views on these records. Based on this data, Chapter 3 introduces a four-step framework to detect and describe modifications to metadata records. We start at very local modifications and group them together to larger units. We demonstrate the feasibility of the framework by applying it to several real-world data sets.

The following two chapters will present different applications of this model. In Chapter 4, we consider modifications which correct defective data. We describe how defect corrections can be extracted from the set of all modifications. From these

data, we create test collections for name-related defects. We analyze the properties of the test cases in the collections and discuss implications for defect detection algorithms. In Chapter 5, we will present two studies based on historic data from DBLP. First, we will show how the thematic coverage of the collection changed over time. We then discuss how users contribute to DBLP.

This work has no explicit related work chapter. Instead, the related work is discussed in the individual chapters.

1.4 Prior Publications

Parts of this thesis have already been published elsewhere, including ideas, results, tables and figures. This work is partially based on the following publications:

- Florian Reitz and Oliver Hoffmann. *Learning from the Past: An Analysis of Person Name Corrections in DBLP Collection and Social Network Properties of Affected Entities.* In ASONAM 2010: Odense, Denmark. [RH10b] And the extended version published in volume 6 of Lecture Notes in Social Networks, 2013. [RH13]
- Florian Reitz and Oliver Hoffmann. *An Analysis of the Evolving Coverage of Computer Science Subfields in the DBLP Digital Library.* ECDL 2010, Glasgow, UK. [RH10a]
- Florian Reitz and Oliver Hoffmann. *Did They Notice? - A Case-Study on the Community Contribution to Data Quality in DBLP.* TPDL 2011: Berlin, Germany. [RH11]

Besides these publications, I created a visualization for DBLP which focuses on the internal time of the collection [Rei10]. I also contributed to a survey which compared test collections for name disambiguation algorithms and created a new one by harnessing the manual work of zbMath [MRR17]. As part of this work, I created and published test collections for the person-name disambiguation problem[Rei18].

Chapter 2

Bibliographic Metadata

Contents

One of the most important tasks of a library is to create and maintain metadata for the documents in its collection. Metadata is an umbrella term for any information that describes the actual documents. Common examples are the title of a document or the names of the creators. Metadata play a central role in retrieving documents (See Figure 2.1). Usually, users enter known information about a document into a retrieval system. The retrieval system evaluates this information against the metadata of the collection. If a document matches the query, most systems render a visual representation of the metadata and present them to the user. These visual representations can also be used for exploratory searches where the user browses the collection (usually on a computer screen) instead of entering queries. Aside from retrieval and presentation, metadata plays an important role in interlibrary communication, for example, inter-library loaning, and is crucial for archiving and preservation [NIS04]. In most libraries, the creation of metadata is governed by a body of rules and standards. Different libraries require different rules. For example, regulations to create metadata for books are of little use for a library that collects photographies.

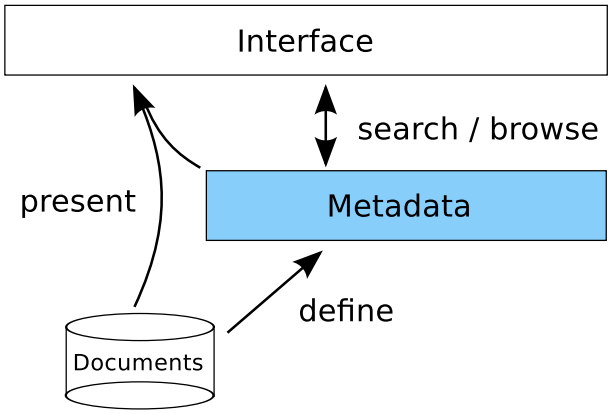


Figure 2.1: Simplified structure of a digital library.

Metadata is not stable over time. As we have seen in the previous chapter, the descriptive information for a document might change. Possible reasons include the correction of errors or that additional information became available such as the birth year of an author. For most applications – in particular, the retrieval task – it is sufficient to use the most recent metadata. The rationale behind this is that only the best possible data should be used. As all modifications somehow aim at improving the data quality (assuming that there are no malicious editors), the new data is expected to be better than the old data. However, we will see that *historical* metadata can be a valuable source of information. In preparation for this thesis, we examined if any of the common metadata standards provides guidelines on how to handle or retain historical metadata. The very popular AACR2 [GW78] (in all versions) and its successor RDA [AAoLP13] ignore modifications completely. The same is true for RAK [Pop93] and the much simpler Dublin Core [UI01]. However, we will see that historical data is available for many projects. But as traditional metadata schemata cannot represent it, it is often provided in an erratic form that needs preprocessing.

In this chapter, we discuss the data on which we will base our study of modifications to metadata. As we do not want to limit ourselves to a single source, we have to find a way to deal with the different formats and content types of existing data. It is not feasible to transform all data sources into one of the commonly known metadata definitions. This is algorithmically very difficult and we will see that we need only a small amount of the metadata definition anyway. Instead, we introduce more general representations for bibliographic metadata (Section 2.1). We then show how these representations can be generalized to store temporal information (Section 2.2). The primary problem of this work was to obtain sufficient historical information. In Section 2.3, we discuss possible data sources and their most relevant properties. We also present the data sources we used in this work.

2.1 Static Metadata

The metadata of a collection is usually provided as a set of records. Each record contains all the descriptive information that is available for a specific document (see Section 2.1.1). Most applications based on metadata require that all records in a collection have some common properties. For example, a query system must know where in the records it can find the document's title. This is easy to achieve if the title is always referenced by the same identifier. To ensure uniformity of the records, libraries use a body of guidelines that the metadata curators must follow. Ideally, two persons would create identical records for the same document. These rules and regulations form the **metadata definition**. According to Lei, Teng and Qin [ZQ08, pp. 15] a metadata definition consists of four different parts. The **data structure** definition defines the data elements that make up the record. For each element, it provides a unique denominator. For example, the author of a document is denoted by the element `creator`. If there are multiple authors, the data structure specifies if they are placed in a single creator element or if there is an element for each author. An element can consist of sub elements (see below). In this case, the data structure definition names the permitted sub elements. The **content guidelines** describe the data that is stored for each element. Among others, it describes where to obtain information, how to handle conflicting data and how to normalize values. For example, a guideline might specify that the last name of an author is given before the first name and that all name parts from the document title should be used. These rules can be quite specific. For example, RAK [vL89, §342.3] specifically regulates how to handle authors who are antipopes. In some cases, the values generated by the content guidelines are not sufficient and additional **data value** definitions are necessary. For example, if an author uses different aliases, a central register is required to map the names. These registers are called authorities. Another example for a data value definition is the Library of Congress Subject Heading database [Lib86]. It contains normalized keywords that can be used to describe the subject of a document. A data value definition can also include references to standards, for example, ISO 639-x for language names and abbreviations and ISO 8601 to format date strings. Many libraries exchange records with other organizations. This is important, as many institutions are not able to create a sufficient number of records on their own. Exchanging records requires that all partners agree on common (or at least convertible) metadata definitions. However, it is also necessary to specify technical aspects of the exchange. The **data exchange** regulations define aspects like data transit protocol encodings, access permissions and remote queries.

There are a number of existing standards for different parts of the metadata definition that libraries can use. Table 2.1 gives some popular examples. Some standards cover multiple aspects of the framework proposed by Lei Teng and Qin. For example, MARC21 provides a structural definition but is also a data exchange format. RAK provides a list of language names that could be replaced by value definitions such as ISO 639-x. If a standard is used, it is often necessary to adapt it to the local requirements [Mil11]. Foulonneau and Riley point out that two libraries collecting similar

Table 2.1: Overview on the elements of a metadata definition.

category	function	examples
Structure	• define required and optional data elements • state quantity of elements • structure data elements	MARC21, DublinCore
Content	• define the content of data element • ensure consistency of data elements	AACR2, RDA, RAK
Values	• provide predefined lists of values • named entity authority • define syntax for certain data types	MARC authority format, LOC subject headings, ISO 639-2
Exchange	• Make data transferable	MARCMXL, MODS, OAI-PMH

content might need to use different metadata based on the requirements of their users and the available funding [FR08, cap. 7]. As we will see in Section 2.3, some collections use self-made metadata definitions. The consequence is that two libraries can create very different metadata records for the same document. Example 2.1 shows two records for the same document ([vL89]) created by the German National Library (DNB) and the Library of Congress (LOC). The most significant differences are caused by the exchange format (DNB: a MARC based text format, LOC: the XML-based MODS). We can also see that the content guidelines of DNB (RAK) make use of abbreviations for standard terms, for example, *hrsg.* for *Herausgeber* (publisher) while LOC (AACR2) does not. Also note that *(RAK)* is part of the title from the Library of Congress. The German National Library lists this string in a separate subfield.

Example 2.1: Two bibliographic records for [vL89]

DNB: RAK, Tablelisting via MARC21

```
245 00 |a Regeln fuer die Alphabetische Katalogisierung |b (RAK) |c
    [hrsg. vom Bibliotheksverb. d. Dt. Demokrat. Republik, Komm. fuer
    Katalogfragen. Red. Bearb. u. Reg.: Elisabeth Lotte von Oppen]
```

LOC: AACR2, MODS

```
<titleInfo>
  <title>Regeln fuer die alphabetische Katalogisierung (RAK)</title>
</titleInfo>
<titleInfo type="alternative">
  <title>RAK</title>
</titleInfo>
<note type="statement of responsibility">
```

[herausgegeben vom Bibliotheksverband der Deutschen Demokratischen Republik, Kommission fuer Katalogfragen ; redaktionelle Bearbeitung und Register, Elisabeth Lotte von Oppen].
</**note**>

Like metadata records, the metadata definition can also be subject to change. Most standards are updated once in a while or a library requires modification of a specific rule. It is also possible that an entire part of the metadata definition is replaced by another standard. These changes can directly affect the metadata records. Therefore, we would like to consider them as well when studying the history of a collection. Unfortunately, this was not possible within the scope of this work. Metadata definitions are composed of text documents which themselves are difficult to analyze automatically. They are complemented by work practice and in-house rules, which are either not documented at all, or their development is not documented or this documentation is not publicly available. For none of the data sets we use in this work, it was possible to obtain sufficient information on the development of the metadata definitions. However, modifications to the underlying rulebooks eventually manifest in the metadata and can be detected. In Section 3.4, we describe briefly how the records themselves can be used to identify such development. Like metadata, it is possible that documents are modified as well. Some documents are subject to natural change, for example web sites. However, even a printed book can be modified, for example, if it is damaged. Printed documents are not available for this study. Analyzing modifications to digital documents requires extensive knowledge on different file formats and a thorough content analysis. This is well outside the scope of this work. Like for the metadata definitions, it is possible to derive a partial view on state changes of documents from the metadata.

As a first step, we now define two generic representations of metadata that can hold data created by any metadata definition.

2.1.1 Metadata Record

The metadata record is the digital equivalent to the paper-based library index cards we saw in Chapter 1. In most approaches, all metadata available for a document is gathered into a single record. As Lei et al. point out:

> The metadata record is considered the basic unit of management and exchange and reflects the tradition of librarianship. ([ZQ08] page 149)

In this work, we assume that records are independent from one another. This means: they might refer to each other but a single record can be interpreted without consulting other records. This approach is simplistic. Documents often have a complex

relationship that is reflected in the metadata. The FRBR [IFL98] standard, for example, differentiates between four types of documents: *work*, *expression*, *manifestation* and *item*. Instances of these object classes are ordered in a hierarchy. A work can be realized through expressions (e.g. in different languages). An expression can be embodied in manifestations (e.g. editions) and so on. However, when delivered to the users, the data are still merged into a single record that fits to the level of detail the user is looking at. In this work, we attempt to analyze different digital libraries that use very different metadata definitions. To accommodate all data, we use a very simple record structure.

Definition 2.1 (Record): Let d be a document that is described by a set of data elements E. Let id be an identifier that is unique within the examined library. We call $r := (id, E)$ a **record (for d)**.

Demanding a unique identifier is a small restriction. Many libraries provide one or more identifiers in their records. Examples are the *International Standard Book Number (ISBN)* for books and the *Digital object identifier (DOI)* for arbitrary documents. There is also often a project specific numbering used for administrative purposes. As we study the development of a library over time, we prefer stable identifiers, i.e., if a record is identified by id at time t it should be identifiable by the same identifier through its whole life time. However, not all data sets provide such identifiers. In Section 2.3, we discuss the problem in more detail. For now, we assume that identifiers are stable over time.

The data elements (often called **fields**) consist of a key and a value. The key is a string defined in the data structure definition. We assume that all information stored under the same key denotes the same type of information. For example, all elements with the key `creator` denote a person or organization responsible for the document. The value can be a simple data point (i.e., a string). However, values can also have a complex structure that is a combination of simple data and nested elements. Consider the following example:

Example 2.2: A simple record

```
title : Learning from the Past
creator: [ Florian Reitz
  mail: reitzf@uni-trier.de
  id: a1234   ]
pages: 150
themes: Digital Libraries, Time
```

The values of the elements denoted by `title`, `pages` and `themes` are strings. Most metadata standards prefer the type string over more specific data types like integer

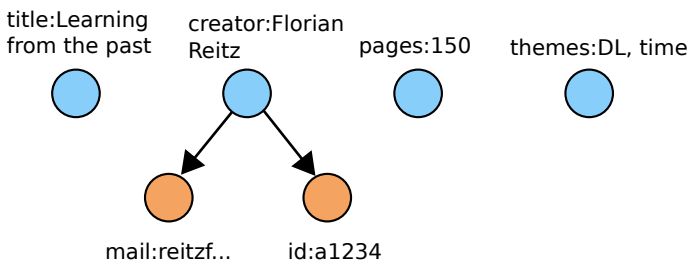
or float because it provides more flexibility to squeeze in atypical documents. For example, the element denoted by `pages` usually contains a number but it also might contain values such as *c 150* or *150–160*. `title` and `pages` store a single data point while `themes` stores a list of topics. Many metadata definitions describe the use of lists, in particular how to handle the delimiters.

> ***Definition 2.2 (value domain):*** Let k be a key in a specific collection. We call the sets of all values that can be associated with that key the **domain of k** and write dom_k.

We consider elements of a list of values as independent members of the domain. While all domains consist of strings, they can differ significantly. In Example 2.2, the domain of the key `creator` consists of personal names like *Florian Reitz* while most elements of `pages`' domain are numbers. Domains are highly specific to the library. Consider the domain of a `creator` key for a library that collects documents from traditional Indonesian authors and the domain of the same key for a library on documents from Europe. While many names in the Indonesian library will consist of a single name, most European names will have two or three components. In praxis, value domains can contain values which are not related to their key. This can be caused by wrongly assigned data (e.g., the year element contains the name of an author) or unexpected use cases that needed to be accommodated.

Some data elements have a more complex structure. Consider the element `creator` from before. Beside the name, it also contains two nested elements (`mail` and `id`). Nested elements are often used to provide additional information for the value. However, in some metadata definitions nested elements are used for grouping elements with similar content. For example, all `creator` elements of a publication are again nested in an element with the key `creators`. We discuss this problem further in Section 2.5. For now, we assume that nested elements cannot be reasonably interpreted in the context of the record without their parent. *mail*, for example, could become an independent element but we would not be able to tell to which author it belongs. If we consider data elements as nodes and nesting as edges, we obtain the following forest as representation for Example 2.2.

> ***Example 2.3:*** Tree representation of Example 2.2.



We use the tree-based view for the formal definition of a data element:

Definition 2.3 (Data Element): Let k be a key with dom_k and $v_1, \ldots, v_n \in dom_k$. We call $e := (k, v_1, \ldots, v_n)$ a **key-value pair**.

Let P be a multi set of all key-value pairs of a record and let T_P be a forest of trees where each element $p \in P$ is represented by the node n_p. Let t_p be the subtree of T_P that is rooted in n_p and let $parent_p$ be the parent of n_p in T_P.

We call $e := (p, parent_p, t_p)$ a **data element**. If $parent_p = \text{null}$ we call e a **head element**, otherwise, we call it **sub element**. If $|t_p| = 1$ we call e a **simple element**, otherwise we call it **complex**.

For a data element e, we call the ordered sequence of ancestors of n_e (starting in the root of the tree that contains n_e) the **top-path** of e.

The forest can be of arbitrary depth. However, we found very few examples where records contained elements with a depth greater than three. Definition 2.3 allows list of values $(v_1, \ldots, v_n)$ which is also rare in the data sets we analyze in this work.

Metadata records are often represented as lists of elements similar to Example 2.1 (page 12). This imposes an implicit order, for example, element `title` appears before element `pages`. Usually, this order is not relevant and Definition 2.3 does not provide an order for sub elements. However, in some cases the sequence of elements has semantics in its own right. For example, if multiple `creator` elements are present, the metadata definition might demand that they are ordered as they appear on the document. In this case, changing the order of these elements would change the content of the metadata record. We use a mechanism similar to XPath to identify elements for which we want to preserve the order.

Definition 2.4 (Axis): Let K be a set of keys and $a_i \subseteq K$ for $1 \leq i \leq n$. We call an ordered sequence $A := a_1, \ldots, a_n$ of a_i and **axis** of a record. We write $A = a_1/\ldots/a_n$.

Let e be a data element with top-path $P = e_1, \ldots, e_m$. We call e **identified** by an axis $A = a_1, \ldots, a_n$ if $m = n$ and $\forall p_i : key(p_i) \in a_i$ for $1 \leq i \leq n$.

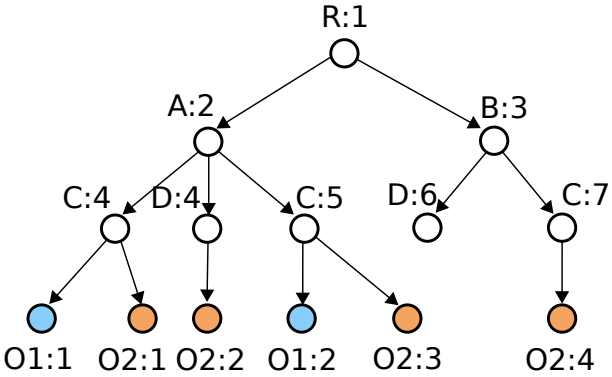
An axis can group elements with different keys. Based in the sets selected by the axes, we define order groups:

Definition 2.5 (Order Group): Let $\mathcal{A} := A_1, \ldots, A_m$ be axes and i a unique identifier then we call $R = (i, \mathcal{A})$ an **order group**.

The **scope** of R is the set of all data elements which are identified by at least one axis in $\mathcal{A}$.

The elements in the scope of an order group are numbered consecutively according to the order in which they appear in the record. That number is added as an additional sub element to the data element. A data element can be member of several order groups. Example 2.4 shows a record with two overlapping order rules. The identifier of the order group is used to distinguish the different rules. Each node represents an element, annotated by $k : v$ where k is a key and v is a value. We will use this notation for the remainder of this work. Axis $\{R\}/\{A\}/\{C\}$ identifies the data elements $(C : 4)$ and $(C : 5)$. The elements with keys $O1$ and $O2$ denote the inserted order information.

Example 2.4: $S = (1, \{\{R\}/\{A\}/\{C\}\})$,
$T = (2, \{\{R\}/\{A\}/\{C, D\}, \{R\}/\{B\}/\{C\}\})$



2.1.2 Metadata Graph

Metadata records can contain links to other records. For example, MARC21 provides a `version_of` field which contains the identifier of another record. However, records are mostly independent data objects. This simplicity hides information which can be important to make sense of metadata. Consider the following records of scientific papers:

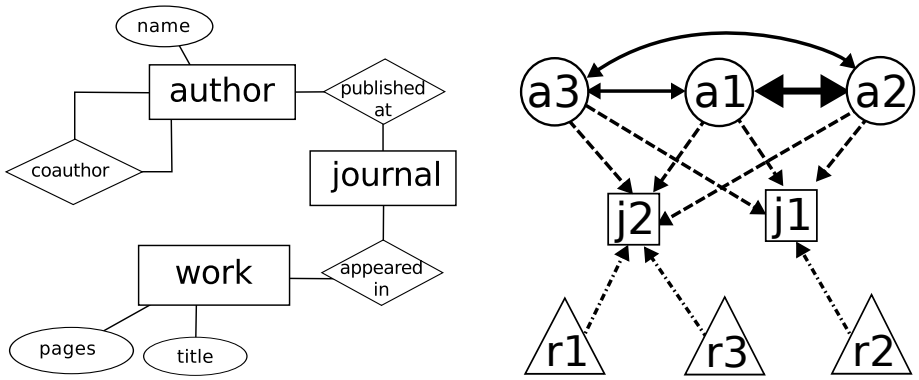
Example 2.5:

r_1 `creators: a1, a2, title: t1, journal: j2 , pages: 10`

r_2 `creators: a1, a2, a3, title: t2, journal: j1, pages: 25`

r_3 `creators: a3, title: t3, journal: j2, pages: 10`

None of these three records references one of the others. However, we might think that r_1 and r_2 are related because some of the authors of r_2 are authors of r_1. While r_1 and r_3 are not in a *shared author* relation, they are related because the papers were published in the *same journal*. The shared author relation in particular has been used in many approaches to find defective metadata in digital libraries. See the surveys of Elmacioglu and Lee [EIV07] and Ferreira et al. [FGL12] for an overview. We discuss this in more detail in Section 4.3.1. The entities and their relations with each other can be modeled by a simple entity-relationship diagram (see Figure 2.2(a)). In the example, we identified three types of entities: *author*, *journal* and *work*. They are connected by three relations. The selection of entities and relations depends on the planned study. In the example, we did not include a



(a) ER-Model (b) Instance

Figure 2.2: An ER-model and a matching graph instance (from Example 2.5). The edge thickness represents the weight of the relation.

relation based on the number of pages. This is because we assume that this relation provides little additional insight. However, in a study that analyzes specifics of short and full papers, this relation might be relevant.

Based on this model, we can define a graph-data structure for our data set:

> ***Definition 2.6 (Multi-layered Graph):*** Let $\mathcal{E}$ be a set of entity types and for $e \in \mathcal{E}$ let E_e be a set of instances. Let $\mathcal{R}$ be a set of unique identifiers.
>
> For $a, b \in \mathcal{E}$ with E_a and E_b, we call $r \subseteq E_a \times E_b \times \mathbb{R}^+ \times \mathcal{R}$ a **relation**. For a relation $r := (e_1, e_2, w, r)$ we call e_1 **source**, e_2 **target**, w the **weight** and r the **relation type** of r.
>
> Let R be a set of relations. We call
>
> $$G = (V := \bigcup_{e \in \mathcal{E}} E_e, R)$$
>
> a **multi-layered graph** with **nodes** V and **edges** R.
>
> Let $\bar{\mathcal{R}} \subset \mathcal{R}$ and let $\bar{R} := \{r := (\cdot, \cdot, \cdot, r) \in R : r \in \bar{\mathcal{R}}\}$. We call $G_{\bar{R}} := (V, \bar{R})$, the **projection** of G through $\bar{\mathcal{R}}$.

In this work, we will use the terms *graph* and **network** interchangeably. All edges are directed. However, for convenience, we use the term **undirected edge** for a pair of two reciprocal but otherwise identical edges. Each edge r is weighted by a positive number $w(r)$. In this work, we assume that small weights indicate weak relations and large weights strong relations. For example, in Figure 2.2(b), we used the number of common publications as a weight for the coauthor relation. G is a multi-layered graph where the maximum number of edges between two nodes is bounded from above by the number of relations, i.e., for each relation and for each

pair of nodes there can be one edge. Edge weights do not stack, i.e., if for a pair of nodes (a, b) there are two edges with weights w_1 and w_2, the nodes are not connected by a relation with weight $w_1 + w_2$.

We can use graphs to study structures in data sets. A central aspect of a structure is the distance between entities. In our model, we assume that nodes connected by a strong edge (i.e., an edge with a higher weight) are *close*. We use the reciprocal of the weight as distance measure. Depending on the application, this might require normalization of the weights. We discuss this point together with the applications in Chapters 4 and 5.

Definition 2.7 (Path, Distance): In $G = (V, R)$ we call $n, m \in V$ **connected** if there is a sequence $p := (s_1, t_2), \ldots, (s_k, t_k)$ with $(s_i, t_i) \in R$ for $1 \leq i \leq k$ so that $n = s_1$, $m = t_k$ and $\forall 1 < i \leq k : t_{i-1} = s_i$.

We call p a **path** from n to m. The **length** of a path p is $l_p := \sum_{(s,t)\in p} \frac{1}{w((s,t))}$.

Let $n, m \in V$ and $\mathcal{P}$ be the set of all paths between n and m. The **distance** between n and m is defined as:

$$dist(n, m) := \begin{cases} 0 & \text{if } n = m \\ \infty & \text{if } \mathcal{P} = \emptyset \\ \min_{p\in\mathcal{P}} l_p & \text{else} \end{cases}$$

A path can combine edges from different relations. This is not always desired. We call a path **pure** in R_i if it consists of edges from R_i. Similarly, we can define the pure distance between nodes. An application of the distance is to define the vicinity of an interesting node. This is often used to reduce the amount of data in order to apply a computationally expensive algorithm.

Definition 2.8 (d-neighborhood): Let $d > 0$ and n be a node in a graph $G = (V, E)$. Let $V_n := \{m \in V | dist(n, m) < d\}$ and $E_n := \{e := (s, t, \cdot, \cdot) \in E | s \in V_n \wedge t \in V_n\}$.

We call the graph $G_n := (V_n, E_n)$ the **d-neighborhood** of n.

Most of the graphs we study in this work are scale free. This means that we can find a large subset of nodes (the *giant component*) which are all connected with each other. This component is a *small world*, i.e., the average distance between two nodes is small. For example, Newman [New04] analyzed the properties of graphs that are defined by the collaboration of scientists. For several different data sets, he found giant component sizes between 57.5% and 92.6% while the average distance (with an edge weight of 1) is between 4.0 and 9.7. A consequence is that even small increases to d can dramatically increase the size of the neighborhood.

2.2 Dynamic Metadata Models

In this section we discuss how to enrich the static metadata models with information on their internal development, i.e., information on the history of the metadata in a specific collection. Is is important to consider the properties of historical data that we can obtain from projects. We will see later that historical information is difficult to get by and that is is often incomplete in different ways. We decided to use an observation-based framework that can handle these deficiencies as good as possible. As before, we discuss a record-based and a graph-based approach.

2.2.1 Dynamic Metadata Record

A dynamic metadata record represents the modifications to a metadata record over time. In this work we use a simple discrete temporal model:

> ***Definition 2.9 (Temporal Domain):*** Let $\mathcal{T}$ be a set of time points and let $<_\mathcal{T}$ be an *earlier than* order on $\mathcal{T}$. We call $\mathcal{TD} := (\mathcal{T}, <_\mathcal{T})$ a **temporal domain**. We call a sequence $T := t_0 <_\mathcal{T} \cdots <_\mathcal{T} t_k$ a **time line** in $\mathcal{TD}$.
>
> Within a timeline T, a **closed interval** $[t_s, t_e]$ is a set of time points with $[t_s, t_e] := \{t \in T | t_s \leq_\mathcal{T} t \leq_\mathcal{T} t_e\}$. (t_s, t_e), $(t_s, t_e]$ and $[t_s, t_e)$ are defined accordingly.

Within a temporal domain, let $r\langle t\rangle$ represent the **state** of a record r at time point t. For now we simply define the state as the set of data elements stored in the record. Two states of the same record are identical if they contain the same data elements. If r is not part of our digital library at time t, we write $r\langle t\rangle = \emptyset$. Over time, the state of a record might change. If we compare the state before and after the change, we can identify a set of newly added data elements e_+, deleted data elements e_- and modified data elements $e_{\neq}$. (we discuss the differentiation between add/remove and modify in Chapter 3). These sets characterize the event that caused the modification.

> ***Definition 2.10 (Event):*** Let $t \in \mathcal{T}$ and e_+, e_- and $e_{\neq}$ as described above. If at least one of the e_+, e_-, $e_{\neq}$ is not empty, we call $e := (t, r, e_+, e_-, e_{\neq})$ an **(modification) event** for record r.

For each event, we obtain a new state. If we can capture all events, we obtain a compact description of the history of a record.

Definition 2.11 (Event Timeline, Revision, Event History): Let $e_1, \ldots, e_k$ be the temporarily ordered sequence of all events that affect record r.

We call timeline $t_1 <_{\mathcal{T}} \cdots <_{\mathcal{T}} t_k$ in $\mathcal{TD}$ that contains all time points at which these events occur an **event timeline** for r.

Let t_∞ be a time point in $\mathcal{T}$ with $t_k <_{\mathcal{T}} t_\infty$. We call

$$r_i := \begin{cases} (r\langle t_i \rangle, [t_i, t_{i+1})) & \text{if } 1 \leq i < k \\ (r\langle t_i \rangle, [t_i, t_\infty)) & \text{if } i = k \end{cases}$$

the **i-th event revision** of r.

We call the sequence $\mathcal{H}_r^E := r_1, \ldots r_t$ the **event history** of r.

The event history is complete if we assume that all modifications are captured in events. It is also minimal with respect to the number of revisions as we do not allow *empty* events (events without modifications). The events must also have different timestamps so there cannot be two events at the same time.

The straightforward definition and the compactness would make the event history a good starting point for our studies. However, the assumptions we made on the available data sets are not realistic. In particular, we assume that an event can be detected in the moment in which it occurs and that we can monitor all events that affected a record so far. We will see in Section 2.3 that many sources for historical metadata do not fulfill these assumptions. To account for these shortcomings, we need to relax our model in several ways.

Consider a digital library which stores copies of its metadata records each night. These copies are retained and we have to use them to reconstruct the metadata development. Because of the nature of the data source, the time that elapses between two event time points is bound from below by 24 hours. If a record is modified at two different times but on the same day, we have no way to differentiate those modifications. To model this problem, we assume that the history of a record is determined by an observer who has no knowledge of the event timeline. At certain points of time, the observer examines the state of record r. The observations define their own temporal timeline.

Definition 2.12 (Observation Timeline): We call a timeline $O := o_1 <_{\mathcal{T}} \cdots <_{\mathcal{T}} o_m$ in $\mathcal{TD}$ that consists of the time points of observation the **observation timeline**.

Different observers can have different observation timelines. However, the properties of the underlying data set strongly affect the observer and set bounds to the observation timeline. Usually, the event timeline (e.g. the actual changes) and the

observation timeline are not identical. In particular, the observation timeline is far more sparse than the event timeline.

Let o_A and o_B be the time points of two consecutive observations. Assume that for a record r, we find $r\langle o_A\rangle \neq r\langle o_B\rangle$. As above, we can model the difference by the sets of added, removed and modified data elements. We obtain an **observed event** $\bar{e}$. This event might group several actual events which occurred between the observations.

> ***Definition 2.13 (Event Group):*** Let $E := e_1, \ldots e_k$ be modification events that affect record r. Let $t(e_i)$ for $1 \leq i \leq k$ be the time point of the event. Let o_A and o_B be two consecutive observations.
>
> We call $E_B := \{e \in E : t(e) \in (o_A, o_B]\}$ the **event group** of observation o_B.

The observed event represents all events in the event group of the observation. If the event group is empty (i.e., there is no event between the observations), we do not observe an event. If the event group contains a single event e, the observed event is identical to e except that we register it at a later time. If more than one event is in the event group, they are merged into a single observation.

> ***Definition 2.14 (Masked Event):*** Let o_A and o_B be two consecutive observations. Let E_B be the event group of o_B for record r.
>
> If $|E_B| > 1$ we call the events in E_B **masked**.
>
> If $|E_B| > 1$ and $r\langle o_A\rangle = r\langle o_B\rangle$, we call the events in E_B **fully masked**.

A fully masked event occurs if a modification is reversed. Assume that two events e_1 and e_2 are grouped in an event group. Assume that e_1 added a data element and that e_2 removed that data element again. Though the state created by e_1 is short-lived (assuming we have a reasonable observation frequency) it might be interesting for this study. For example, it might give insights into what errors are introduced over time and how the quality assurance for modifications works. However, because of the limitations of the observation-based model, these modifications are lost.

Whether events are masked or not is partly determined by the observation framework. In the example of the fully masked event, we assumed $o_A <_\mathcal{T} t_1 <_\mathcal{T} t_2 <_\mathcal{T} o_B$ where o_A and o_B are observations and t_1 and t_2 are times of events. In a slightly different observation framework, the situation might be $t_1 <_\mathcal{T} o_A <_\mathcal{T} t_2 <_\mathcal{T} o_B$. In this case o_A (by comparing it with previous observations) has detected an unmasked event. The same would be true for o_B. For studies based on observed states this means that (1) we will be unaware of some events (2) The set of events that we do not register partly depends on the (often externally imposed) time points of observations. This problem becomes significant if the gap between two observations is large. For some of our data sets, observations are one week apart. For others,

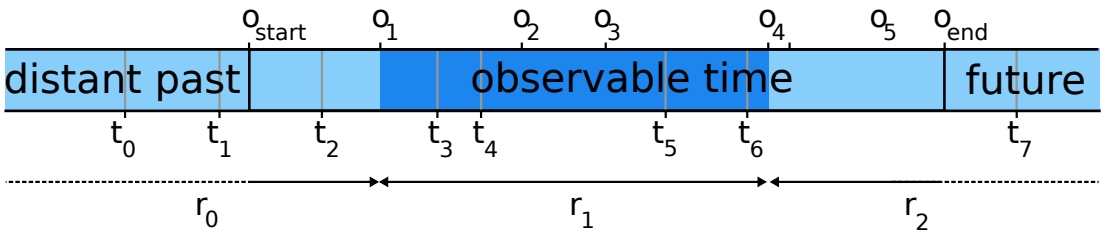


Figure 2.3: Observation of a single record. $t_0 \dots t_7$ denote the event timeline and $o_{start} \dots o_{end}$ the observation framework. In this example, the observations create three revisions r_0, r_1 and r_2. Events at t_3 and at t_4 are fully masked. Events at t_5 and at t_6 are partially masked. The record is primordial and surviving. Events t_0, t_1 and t_7 are outside the observed time frame.

we cannot tell the exact time between observations. Partly masked events are also problematic. While we obtain an observed event in this case, it is possible that single data elements are fully masked. In the example above, the added and then removed data element would remain invisible even if there were other modifications that are observed.

The second problem with the observation framework is that it might not cover the entire lifetime of a record. So far, we implicitly assumed that the first event *creates* the record by adding the first data elements. In reality, it is very likely that some records existed for a time before the beginning of the observation. There are two scenarios: (1) The record existed before the observation started. (2) Some revisions of the record are not available from the data. Assume that a library performs quality assurance steps for each new record. Only after the last step, the record is added to the actual library. It is possible that the quality assurance steps created revisions but that these revisions are not available for us. We have no way to determine if there are missing steps in the second scenario. However, we can handle the first scenario:

> ***Definition 2.15 (Primordial and Surviving Records):*** Let $O := o_1 <_\mathcal{T} \dots <_\mathcal{T} o_k$ be an observation timeline. If $r\langle o_1 \rangle \neq \emptyset$, we call r **primordial**. If $r\langle o_k \rangle \neq \emptyset$ we call r **surviving**

Surviving records pose no particular problem. On the other hand, we do not have any information on the development of primordial records from before the start of the observation. Leskovec et al. [LKF07] found this property in many data sets they studied and coined the term *missing past*. If we start our observation of record r at time o_{start} and we obtain an **initial state** $r\langle o_{start} \rangle$, then we cannot determine how long the record existed and which modifications were made. Primordial records might appear to be more stable than records created during the observation as there was time to correct errors. This must be considered in studies that include these

records. We refer to events that occurred before the first observation as the **distant past** of a collection.

With these limitations, we can define a record history that strongly depends on the observer:

Definition 2.16 (Observed History): Let $\bar{e}_1, \dots \bar{e}_m$ be the temporally ordered non empty observed events in an observation framework bounded by o_{start} and o_{end}. Let $t_1 <_\mathcal{T} \cdots <_\mathcal{T} t_m$ be the timeline that represents the observed events. For $0 \leq k \leq m$ and $t_{-\infty} <_\mathcal{T} t_0$ and $t_m <_\mathcal{T} t_\infty$, we call

$$r_k := \begin{cases} (r\langle o_{start}\rangle, (t_{-\infty}, t_2)) & \text{if } k = 0 \\ (r\langle t_i\rangle, [t_i, t_\infty)) & \text{if } k = m \\ (r\langle t_i\rangle, [t_i, t_{i+1})) & \text{else} \end{cases}$$

the **k-th observed revision** of r. Let $r_k = (s, I)$ be a revision. We call $val(r_k) := I$ the **valid interval** and $state(r_k) := s$ the **state** of r_k.

We call the $\mathcal{H}_r^O := (r_0, \dots r_t)$ the **(observed) history** of r.

With the limitations discussed above, we again obtain a minimal description for the history of a record. Figure 2.3 gives an overview on the time concepts for records. Again, for all time points in the valid interval of a record, the state is identical. If we define t_∞ and $t_{-\infty}$ in a way that they are arbitrarily far in the future or past, we can define the state of a record.

Definition 2.17 (Dynamic State): Let t be a time point with $t_{-\infty} <_\mathcal{T} t <_\mathcal{T} t_\infty$ and let r_i be a revision of r with $t \in val(r_i)$. We call $state(r_i)$ the **state** of r at point t and denote it as $r\langle t\rangle$.

Not all records exist through the whole observation period. It might be absent at the beginning or it might be deleted later. We define:

Definition 2.18 (Record Lifetime): For a record r and an observation framework as above, we call

$$\mathcal{L}_r := \bigcup_{r \in \mathcal{H}_r^O : \text{state(r)} \neq \emptyset} val(r)$$

the **lifetime** of r.

So far, we considered single records. To describe the history of a collection, we combine the histories of the individual records.

Definition 2.19 (Collection set, State): Let R be a set of records that are stored in a collection during an observation interval bounded by o_{start} and o_{end}

For $t \in [o_{start}, o_{end}]$, we call $R_t := \{r \in R | t \in \mathcal{L}_r\}$ the **collection set** at time t.

We call $R\langle t\rangle := \bigcup_{r \in R} r\langle t\rangle$ the **state** of the collection at time t.

Definition 2.19 uses a global observation framework, i.e., the times of observation are identical for all records. This is not a relevant limitation as all data sets we obtained for this study enforce a global observer. However, the definition could be extended to record-specific observers if needed.

We can define a global history for a collection based on the local observed events. Let $E := e_1, \ldots, e_k$ be the sequence of all observed events that affect records in R. Let E_t be the subset of E, that were observed at time t. Each E_t is associated to a specific state of the collection.

Definition 2.20 (Global Observation History): Let $E_1, \ldots, E_m$ be a partition of the observed events, with E_i as described above. Let $\forall E_i : E_i \neq \emptyset$.

With an observation framework as above. Let $t_1 <_\mathcal{T} \cdots <_\mathcal{T} t_m$ be the timeline that represents these event sets. For $0 \leq k \leq m$ and $t_{-\infty} <_\mathcal{T} t_0$ and $t_m <_\mathcal{T} t_\infty$ we call

$$r_k := \begin{cases} (R\langle o_{start}\rangle, (t_{-\infty}, t_2)) & \text{if } k = 0 \\ (R\langle t_i\rangle, [t_i, t_\infty)) & \text{if } k = m \\ (R\langle t_i\rangle, [t_i, t_{i+1})) & \text{else} \end{cases}$$

the **k-th observed global revision** of R.

We call the $\mathcal{H}^O_\mathcal{R} := (r_0, \ldots r_t)$ the **(observed) history** of the collection.

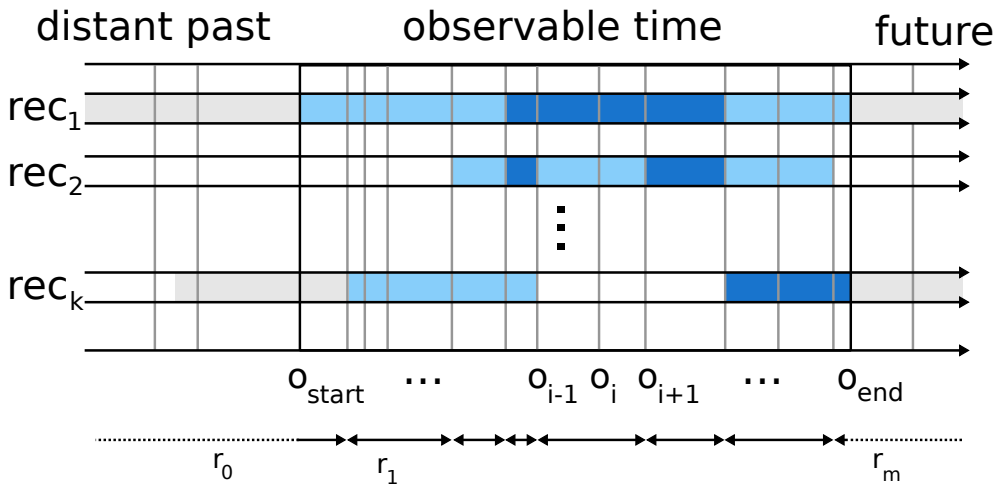


Figure 2.4: The history of a collection derived from the history of three individual records.

Figure 2.4 shows an example of a global history. The concepts of distant past and future translate directly from individual records to the collection. For convenience, we define sets that denote which records are affected by a revision:

> ***Definition 2.21 (Added, Deleted and Modified Records):*** Let R_a and R_b be the collection sets of two consecutive revisions.
>
> We call $R_b^+ := \{r \in R_b | r \notin R_a\}$ **added records**, $R_b^- := \{r \in R_a | r \notin R_b\}$ **removed records** and $R_b^{\neq} := \{r \in R_a | r \in R_b \wedge r\langle a \rangle \neq r\langle b \rangle\}$ **modified records**.

By definition, at least one of R_b^-, R_b^+ or $R_b^{\neq}$ is not empty.

2.2.2 Dynamic Network Structures

We can extend the network-based view using the observation framework described in the previous section. Most concepts translate directly from the record-based view to the graph-based view.

> ***Definition 2.22 (Dynamic Graph):*** For an observation time frame $O := o_1 <_{\mathcal{T}} \cdots <_{\mathcal{T}} o_m$ let G_{o_i} be the graph at time o_i.
>
> We call
>
> $$\mathcal{G}_O := G_{o_1}, \ldots, G_{o_m}$$
>
> a **dynamic graph** in O.

Based on the dynamic graph, we can define revisions as well as a revision-based history similar to the record-based view. However, as there is no truly local information in a graph (unlike for records), there is no straightforward local history. The following operations can change a graph: *add/remove node* and *increase/decrease relation strength*. The later operations include creation of new edges and removal of old edges. Node and edge lifetimes can be defined similar to the record lifetimes.

Dynamic graphs are more difficult to handle than the collection history derived from the record-based view. With records, we just need to store the local history of each record. From this, it is easy to derive the global history. The information in metadata graphs is not local. A single event can directly or indirectly affect a large number of nodes and edges. Metadata graphs are also very large. Even determining if two succeeding graph states are different can be a challenge. Because of this, it is infeasible to use dense observation frameworks. In this work we use dynamic metadata graphs in one of the following ways: (1) As full dynamic graphs with a sparse observation framework. E.g., we only consider one graph per observed year. (2) We consider small sections of the graph.

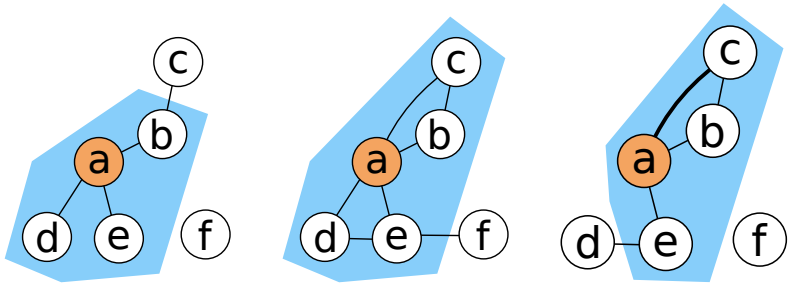
For a number of studies, we are not interested in the global history of a graph but only in changes to the neighborhoods of certain nodes. Usually, these nodes are selected because they have a property that makes them particularly interesting for a certain question.

> ***Definition 2.23 (Dynamic Neighborhood):*** Let $\mathcal{G}_O := G_{t_1}, \ldots, G_{t_m}$ be a dynamic graph and let n be a node in n. For $1 \leq i \leq m$ let $A^d_{t_i}(n)$ be the d-neighborhood of n in G_{t_i}
>
> We call the dynamic graph $A^d_O(n) := A^d_{t_1}(n), \ldots, A^d_{t_m}(n)$ the **dynamic neighborhood** of n

Figure 2.5 shows an example of a dynamic neighborhood for three revisions. Computing the neighborhoods is still time-consuming as their growth can be affected by any part of the complete graph. Still, the pre-selection of interesting nodes makes working with them feasible for smaller d. Definition 2.23 assumes that a specific node n can be identified in each graph of the dynamic graph. In practice, this is not guaranteed as we demand no unique and stable identifiers for nodes and edges.

2.3 Categorizing Sources for Historical Metadata

As part of this work, we attempted to find digital libraries which provide sufficient historical metadata to perform proof-of-concept studies. As pointed out before, many projects do not retain *old* data or are reluctant to share it. If data is available, it is often incomplete and in a format that cannot be used directly. We adapted the metadata model, so it can handle the most significant problems, e.g., the masked events. However, a number of factors can still reduce the quality of results or make



(a) revision 1 (b) revision 2 (c) revision 3

Figure 2.5: The dynamic neighborhood of node a in three revisions. Edge direction is omitted. Note that nodes can be removed from the neighborhood without deleting them from the global graph (node d).

certain studies impossible. In this section, we present a framework that used nine dimensions to describe how suitable a data set is for historical analysis. In section 2.4, we will apply this framework to real world data sets which we used in our study.

The framework for the categorization of historical metadata in digital libraries consists of nine aspects. The first three aspects are related to temporal aspects of the data set ($\mathcal{T}$). The other aspects are attributed to the tracking and retaining of historical revisions ($\mathcal{H}$). Note that the categorizations are specific for an export format of a collection, i.e., if historical data can be obtained in different formats, it is possible that each source has a different categorization. We also found out that properties of a collection can change rapidly.

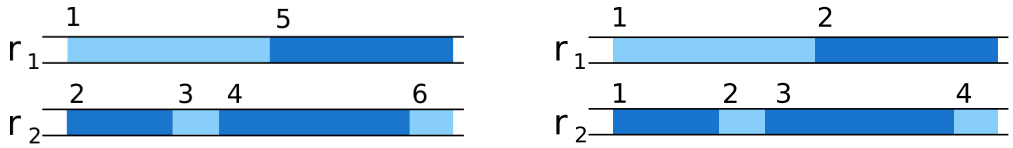
For each aspect, we give a name and a set of possible parameter values.

Type of clock ($\mathcal{T}_{\text{clock}}$)(*real time/logic*) In what way are the timestamps for historical data provided?

For most collections, we cannot perform observations on our own but must reconstruct them from historical data. This is because live observation is often not possible and reconstruction provides data from a longer period of time. When reconstructing the historical states of records, we need to use the clock that is used by the project.

real time: Many projects use a variation of real timestamps, i.e., the timestamps that can be related to a specific date and time. Note that the granularity of those timestamps is often lower than theoretically possible. E.g., a project might provide historical data annotated with a nanosecond timestamp. However, the data are created once a night so the actual temporal resolution is a full day. For the test data sets we observed granularities of up to full weeks.

logic: Timestamps can also be created by logic clocks. The clock defines a number of events, e.g., an edit to a record. For every event, the clock ticks, i.e., increases its internal value. In the simplest form, the counter is a number that is increased by one for each event. This value is used as a timestamp. Within the scope of the clock (see $\mathcal{T}_{\text{scope}}$ below) the timestamps create a *happened-before* order. However,



(a) global clock (b) local clock

Figure 2.6: Comparison of a global logic clock and a local clock with a scope limited to a single record. Note global timestamps 1 and 2 actually occurred at the same time. The order in this case is random.

it is not possible to determine the real time elapsed between two ticks of the logic clock. We will see studies that are based on the duration of revisions and the time that certain information remained in the collection. Obviously, those studies cannot be performed on collections with a logic clock.

Clock scope ($\mathcal{T}_{\text{scope}}$) (*global/local*) Is there a global clock that generates timestamps or do subsets of the collection use different clocks. Assume that a library collects records and their history from different sources. It is possible that these sources use different clocks to annotate historical data. In reality, the differences often come from different data storage strategies used by the projects. Assume that project A preserves historical data every hour, while project B does so every week. If data from these projects are gathered, the records from A are created by a different clock than the records from B. Logic clocks can be local in a way that they use a different counter for different sets of records. Figure 2.6 gives an example of two clocks. One uses a single counter for both records. The other clock has an individual counter for each record. A lack of a global clock scope makes it impossible (or at least difficult) to determine temporal correlation between record revisions. Any studies which analyze the order of modifications or the simultaneousness of modifications are limited to data from the same clock scope.

Intrinsic revision order ($\mathcal{T}_{\text{order}}$) (*yes/no*) Is there a reliable order between revisions of the same record? Revisions of a record are ordered by their timestamps. However, in some situations, we encounter data where two or more revisions have identical timestamps. This is the result of real-time timestamps with low resolution, especially when combined with high modification frequency. Some projects provide means to order the revisions anyway. In case we cannot extract an order directly from the data, we apply the following heuristic: Let $\Delta(r_1, r_2) \rightarrow [0, \infty)$ be a function that measures the difference between the states of two revisions. Let further $r_1, \ldots, r_k$ be revisions with indistinguishable timestamps. We order the revisions in a way that the sum of the differences between succeeding revisions is minimal. If multiple orders with minimal delta exist, we choose randomly.

Global history completeness ($\mathcal{H}_{\text{global}}$) (*full/partial*) As pointed out before, most collections will have a missing past, i.e., before a certain point of time, no historical information can be extracted for the records. It is common that historical data sets

only date back some time or are only retained for a certain interval. Studies must consider that the *early life* of primordial records cannot be described. If a complete overview on the lifetime of records is target of a study, primordial records must be considered separately.

Local history completeness ($\mathcal{H}_{\text{local}}$) (*full/local missing past/import*) Is the development of a single record within the observable time frame traceable?

full: Any significant step in the development of the record is available. It is often difficult to determine the early steps of a new record as they are often done outside the revision tracking system. Therefore, it is unlikely to get all revisions even if there is no global missing past or the global missing past is already over.

local missing past: Early revisions can be missing for an individual record, however, once a revision is tracked, all succeeding revisions of this record are tracked as well. Many projects do not add a new record directly to their collection but put it through a quality assurance process. This process can be extensive and result in multiple alternations to the record. As this process occurs outside the collection itself, it might be outside of tracking. A local missing past can also be the result of collections that retain a limited number of revisions per record. Records with an extensive missing local past appear more stable because many changes are hidden.

import: A special case of missing past. As data exchange protocols ignore historical metadata they are usually not transferred when aggregating data. I.e., the past in the source project is lost. While local missing past within the project can be estimated in most cases, this is not possible for imported records. Often, the data handling processes of the source project are not known and not available. It is also difficult to determine the age of the record.

Trigger for historical artifact creation ($\mathcal{H}_{\text{trigger}}$) (event/interval/irregular) To preserve historical data, a collection must create data artifacts that store the old data. There are significant differences in what triggers the creation of such an artifact:

event: A data artifact is created for every modification event directly after the event occurred. The revisions we can construct from these artifacts are timely and the risk of masked events is low. The version control system of Wikipedia is an example of such a system.

interval: At a certain interval, the collection creates a snapshot of its data. These snapshots are usually used for backup and restore purposes. By comparing the records in consecutive snapshots, we can obtain revisions for a record. All events that affect a record during an interval are grouped together. Interval-based artifact creation imposes a strict observation framework. One of the data sets we use has a creation interval of one full week.

irregular: Snapshots of the records are created at arbitrary times. This is similar to *interval*, however, it is more difficult to determine if all snapshots are available or if some are lost. Large variations of the time between two snapshots makes studies of the temporal relation of modifications difficult.

Traceable record identifier ($\mathcal{H}_{\text{ID}}$) (*yes/no*): Does a record retain at least one reliable unique identifier when it is modified? The observation framework demands a unique identifier for each record. However, as historical data is often a side product, identifiers might not be stable over time. If no stable ID is available, a heuristic has to be used which can fail and thus reduce the quality of the historical data.

Deleted records/revisions ($\mathcal{H}_{\text{del}}$) (*yes/no/notification*) Does the historical data contain deleted records and revisions? Deleting data is not a typical operation for digital libraries but can occur, for example, to remove duplicate records or due to legal considerations. Some projects do not include deleted revisions in their data export. A delete can be limited to a subset of revisions of a record. If these revisions are missing, we can get only partial results on the properties of alternations, in particular if a specific property triggers the deletion.

notification: Some projects do not provide the deleted revisions/records but notifications on the deletion. In this case, affected records can be excluded if the effect on data quality is extensive.

Change comments($\mathcal{H}_{\text{comment}}$) (*no/optional/mandatory*) Can changes of metadata be commented by the data curator? A comment can be a free text with an explanation or more structured data like tags from a controlled vocabulary. The temporal framework can store these data for revisions and for data elements. Comments do not affect studies directly but can provide additional insight on the editing process which can be used to check the results from the framework. Edit comments themselves can be study subjects in their own right (see [AKM08] for an example on commit comment analysis in the related field of source code change analysis).

2.4 Data Sets

In this section we will discuss the historical data that can be extracted from a number of real world digital libraries. We will show typical problems with obtaining historical data and how they can be overcome. In Chapters 4 and 5, we will present studies based on some of these data sets.

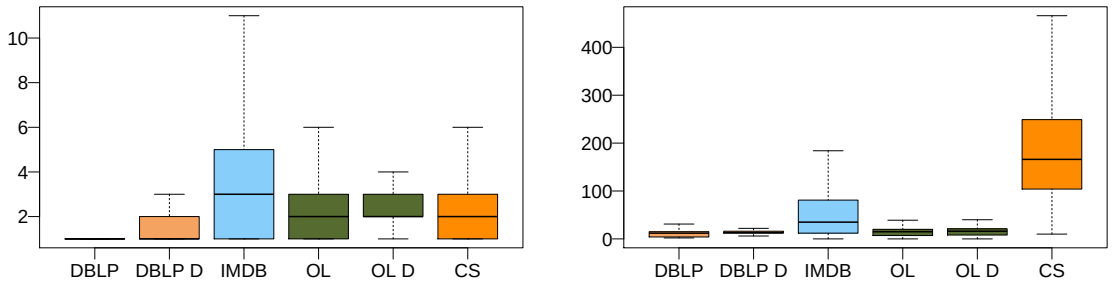
The goal of our data acquisition is to gather historical data for a number of projects from different domains. To achieve this goal, we had to include projects which are generally not considered to be digital libraries. All of the projects we discuss below store descriptive data on (multimedia) documents and use this information to group and search documents. We obtained data from five different projects: *DBLP* and *CiteSeer* (CS) aggregate metadata on scientific publications in the field of computer science while *Open Library* (OL) considers works from all topics. *IMDB* gathers data on films and TV productions. From *Wikipedia*, we created a set of collections based on highly structured article content. DBLP and Open Library contain records which are used for name authority. The change pattern of these records is different from the pattern of the actual records. We created the sets DBLP_D and OpenLibrary_D which contain only records for documents.

Table 2.2: Basic figures on the temporal development of analyzed projects. For Wikipedia only the extreme values are given (see Appendix A.2 for details). *: used in at least 100 different records. † same value as for the complete collection. ‡ dataset from October 2012.

	records				timestamps		
project	**count**	avg_{rev}	**keys**	**keys***	**start**	**end**	**count**
DBLP	4,855,424	1.32	27	24	1999-06	2015-11	5386
DBLP_D	3,148,011	1.47	27	24	1999-06	2015-11	5386†
IMDB	3,447,860	4.09	96	86	2009-03	2012-06	168
CiteSeer	1,388,857	2.20	41	32	n.a.	‡	n.a.
OL	48,439,898	2.47	704	156	2008-03	2012-10	23 Mio.
OL_D	41,399,432	2.66	698	148	2008-03	2012-10	23 Mio.†
WP min	1,717	1.07	20	3	2001-03	2011-02	2,998
WP max	1,033,290	31.54	14,941	563	2011-12	2012-11	2.1 Mio

Table 2.3: Comparison of temporal properties in analyzed projects.

	DBLP(_D)	**CiteSeer**	**OL(_D)**	**IMDB**	**Wikipedia**
$\mathcal{T}_{\text{Clock}}$	real time	logic	real time	real time	real time
$\mathcal{T}_{\text{Scope}}$	global	local	global	global	global
$\mathcal{T}_{\text{order}}$	no	yes	yes	yes	no
$\mathcal{H}_{\text{global}}$	partial	partial	partial	partial	full
$\mathcal{H}_{\text{local}}$	import	missing past	import	full	full
$\mathcal{H}_{\text{trigger}}$	interval	event	event	interval	event
$\mathcal{H}_{\text{ID}}$	yes	yes	yes	no	no
$\mathcal{H}_{\text{del}}$	yes	no	no	yes	notification
$\mathcal{H}_{\text{comment}}$	no	optional	no	no	optional



(a) Number of revisions per record

(b) Number of data elements per revision

Figure 2.7: Distribution of properties for different data sets. Outliers are omitted.

Table 2.2 shows the size of the different collections. For Wikipedia only the size of the largest and smallest collections are given. Table 2.3 shows the properties we obtained with the classification framework we discussed in the previous section. Figure 2.7 shows the size of the record states and the number of revisions for the different collections. Details on the Wikipedia sets can be found in Figure 2.8 on page on 38. Besides the size, there are other important differences which are related to the way historical data is handled. Our modification detection framework equalizes many of these differences. However, some properties of the data must be discussed in more detail as they can influence studies on the internal time of data sets.

DBLP and CiteSeer

DBLP (Digital Bibliography & Library Project)[1] and *CiteSeer*[2] (now renamed to *CiteSeerX*) both aggregate metadata on scientific publications in the field of computer science. DBLP obtains most of its data by crawling web pages and from the publishers directly while CiteSeer uses web crawlers to locate and download papers from the internet. While DBLP aims at adding all papers from a journal issue or conference proceedings, CiteSeer often has only partial information but includes more *gray* literature such as technical reports and theses. There are many documents shared by both collections. To create records, CiteSeer extracts metadata from the documents and the websites they were found on and processes them automatically. However, users can correct entries and provide new data. DBLP relies heavily on manual processes. In general, a CiteSeer record is significantly larger than a DBLP record. In particular, it contains citations of other papers and abstracts. However, the information which is available from DBLP is often more structured and strongly normalized.

Neither DBLP nor CiteSeer provide historical metadata on their web interfaces, but past revisions are retained. Both projects use a very similar data manifestation. The records are stored in small XML-files which are organized in directory trees. CiteSeer apparently uses a system which creates a new file for each edit event. The old file is renamed but retained. These files were kindly provided by Mr. C. Lee Giles in October 2012. The revisions are numbered but we could not determine the change date. We interpret this as logical time with a scope limited to a single record. The XML-files contain structuring elements. For example, all authors are grouped under one *authors* tag. According to our understanding of data elements, we removed those groupings. We discarded a small number of records that were not readable. DBLP creates daily snapshots of the files which are ex post merged with the current data into a *historical directory tree*. The project provides a file called *hdblp* which contains the information from this tree. Old versions of *hdblp* contained revisions which did not differ from the previous ones. This is caused by modifications which were reversed the same day, i.e., fully masked edits. The current version does not contain these revisions. The project has a global missing past which predates all

[1] `https://dblp.org`
[2] `http://citeseerx.ist.psu.edu`

modifications before June 1999. In addition, we can track the creation of records, i.e., the date of the first revision as of October 1995. As metadata is imported and passed through a preprocessing pipeline [Ley09], each record has a local missing past. DBLP has specific records for personal name authority. Their primary purpose is the differentiation of authors with similar names and the provision of information such as an author's web page or affiliation. These pseudo records are smaller and more stable than regular records. We will consider the two types of records separately.

IMDB

The *Internet Movie Database (IMDB)*[3] is a digital library on movies and TV series. The web interface allows users to browse for productions or for persons. Person and production each have their own set of metadata, e.g., runtimes and budgets for productions and biographies for persons.

Every week, the IMDB project provides the current metadata as a nonstandard database dump which consists of 45 separate files. Each file represents a certain type of data, for example, *countries* contains a mapping of movies to their release date in different countries. The structure of these files can be quite complex and the data must be extracted carefully. The purpose of this dump is to create mirror pages. Every week, the information on the mirror should be replaced by the next data version. As all information is replaced, there is no need to ensure stable identifiers or retain older dumps. Productions and persons are identified by their title or name respectively. During the observed time, these identifiers changed several times. For example, 'The Godfather' was changed to 'Godfather, The' and back again several times. We applied a heuristic based on similarities of the title and the data elements of the record. However, the data set has by far the highest number of deleted entities which is because we did not detect renamings. Instead of downloading the complete data each week, users can obtain diff files. These files are available as from June 2009 (older files exist but some diffs are missing so they could not be reconstructed). Starting from the current version of the data files, we worked our way back by reverting all past modifications. The time granularity is one week which is the largest of all collections we study so we assume there is a high number of masked edits. Nevertheless, IMDB has volatile records with an average of 4.09 revisions. We decided to create records for productions and persons which contain the associated information. Most of the time, we use the file name as head element key and associate the primary information with it. All other information is stored in sub elements. Appendix A.2 describes the import in more detail.

Wikipedia

Wikipedia[4] is a crowd-based digital encyclopedia project available in different languages. Users can create new articles or modify existing ones. For each modification,

[3] `http://www.imdb.com`

[4] `http://www.wikipedia.org` data obtained from `http://dumps.wikimedia.org`

the software behind the project creates a new version of the entry. By default, users see the most recent version but older versions remain accessible. Versions for the same entry can be compared and modifications can be reverted if necessary.

Wikipedia provides different types of metadata for each article. For example, articles can have categories which define a (mostly) hierarchical tagging system. The user can search for all entries which have a specific tag. Unlike other digital libraries however, the metadata are not stored in separate records but are part of the article text itself. If a user wants to add a new category, she has to add the name of the category in a specific syntax. I.e., documents and metadata are created alongside by the users.

Previous studies have analyzed the quantity and quality of different types of metadata in Wikipedia entries, particularly the *category system* (tagging and classification) *interwiki links* (multi language support) and *internal link structure*. In this work, we concentrate on a rich but less stable information source, the templates. A template is a code fragment that is part of the article text (see Example 2.6). A template has a unique name (here *Persondata*) and a set of key-value pairs. Values can contain other templates. The template itself defines how the values are rendered when the article is presented to the user. The code from Example 2.6 is visualized as a table containing the biographic data of Donald Knuth[5].

Example 2.6: Personal data infobox of Donald Knuth (source: `https://en.wikipedia.org/wiki/Donald_Knuth` (accessed April 17th 2017))

```
{{Persondata
|NAME= Knuth, Donald Ervin
|ALTERNATIVE NAMES=
|SHORT DESCRIPTION= [[Computer science]]
|DATE OF BIRTH= {{Birth date and age|mf=yes|1938|1|10}}
...}}
```

Some templates are referred to as infoboxes. Though there is no strict definition, infoboxes are templates that create dominant visual elements with a large amount of information. Often, these elements are boxes displayed at the top of an article that sum up significant figures on the article's topic. We will consider these infoboxes as metadata record for an entry and analyze their temporal development. We choose infoboxes because (1) usually, there is at most one infobox of a specific type per article. (2) They have more key-value pairs than average templates. (3) Their visibility creates an incentive to modify them while many templates remain hidden in the entry's text.

We obtained XML database dumps for the English-language Wikipedia (contains revisions up to October 2012), the German-language Wikipedia (October 2012)

[5]The table of this particular template is hidden by default but users can decide to show it.

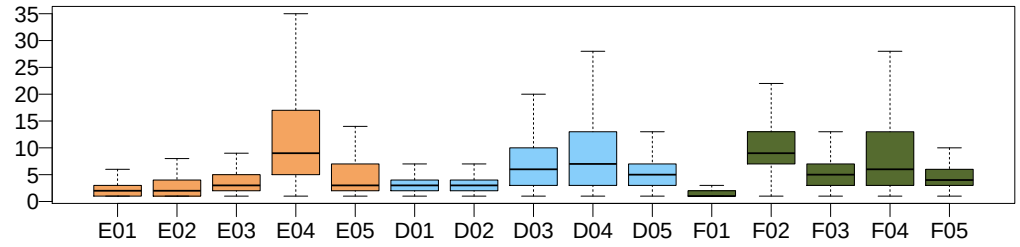
and the French-language Wikipedia (November 2012). At that time, those were the projects with the most articles. The XML files contain full texts of all public versions of an article. They do not contain deleted articles and hidden versions. However, the identifiers of deleted records are listed in a separate file and hidden versions appear as revisions without content. Renaming articles or moving them usually preserves the version history. Some articles are imported from other projects. Import from projects that use the same software can be done in a way that preserved the history of the article. However, a local missing past for a small number of articles is possible.

In a first step, we identified the templates within the article text and filtered out other information. Then we attempted to extract the key-value pairs. Users add and edit infoboxes as plain text. Because there is no direct control of the input, there are a number of deviations from the expected syntax. We attempted to compensate frequent errors and discarded completely unreadable data. We determined the living time of a record as the time between the first and the last appearance of the infobox in an article. Within this interval, we ignore gaps without infobox if they are shorter than one week. In most cases, these gaps are caused by defective edits or vandalism. The first type is usually fixed within a few hours and the second one does not contribute to the development of the record. For each language, we determined the 50 most frequent infoboxes. An infobox can be renamed or addressed by a synonym (called redirect in Wikipedia terminology). If we are sure that two names refer to the same infobox for the entire observable time we map them. The size and update frequency of records differs significantly for different infoboxes. The most significant properties of the selected infoboxes are listed in Appendix A.2. Figure 2.8 shows the numbers of revisions and data elements for the five most frequent infoboxes per language.

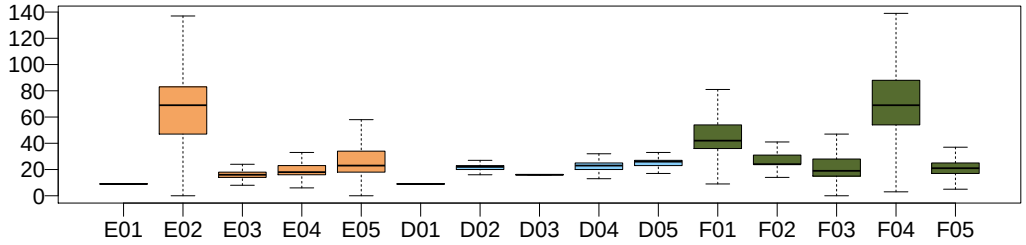
Open Library

The Open Library project[6] is an aggregator for bibliographic metadata, primarily for books. Once a data record is imported into the system, users can edit it. The interface is similar to Wikipedia, however, it strictly separates documents and metadata. Like Wikipedia, each edit creates a new version. Old versions are retained. In addition to the manual edits, there are a large number of automatic modifications, which are logged as well. Open Library provides a complete data dump. Each version is stored in a single JSON record. JSON stores hierarchical key-value data in a format compatible with the JavaScript object specification standard which can be fed directly into our framework. Each record is provided in full but deleted revisions appear to be missing. The dump also does not contain edit comments from users. We analyze the dump from October 31st, 2012. Like DBLP, Open Library has a specific record for each author which is used for name authority. Unlike the previous collections, OpenLibrary features a large set of keys. An examination of the underlying records shows that this is the result of merging different metadata standards without normalizing the keys. Of all collections, Open Library has the

[6] `http://openlibrary.org`



(a) Number of revisions



(b) Number of data elements per revision

Figure 2.8: Boxplots for the number of revisions and data elements of the five most frequent infoboxes per language. See Appendix A.2. Outliers are omitted.

highest number of automated edits. Frequently, they affect the same record, sometimes causing multiple revisions with the same timestamp. However, the data set has a reliable numbering of a record's revisions so we do not need heuristics to sort them.

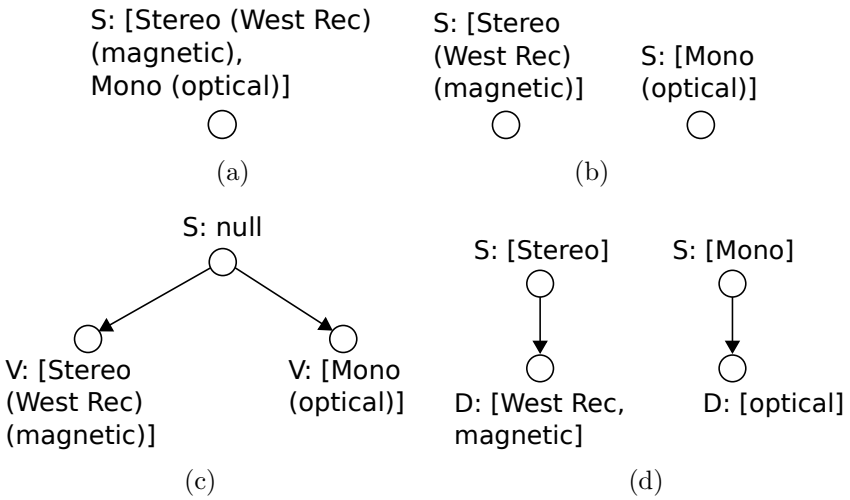
2.5 Threat to Validity: Mapping Bias

Every data set we discussed in the previous section comes with an individual data structure that must be transformed to fit into our detection framework. The mapping of the original format and the result of the transformation can introduce a bias to the detection of modifications. For example, in IMDB, the data we obtained provides very little structure. While the head elements themselves are readily identified, the sub element structure is completely unclear. Consider Example 2.7 that shows an excerpt from the sound-mix data file. Both lines refer to the same Movie M but contain independent information. Below the lines, we can see four different mappings which can be created from this example:

The difference is the number of data elements we use to model the data. This has two aspects: multiplicity of data elements from the same type and complexity of the model. The multiplicity aspect determines if independent data pieces of the same type (here: sound-mix) are modeled as a single data element (a) or as separate data elements (b), (c) and (d). Assume that we add another data point or remove an

Example 2.7:

```
M Stereo  (West Rec) (magnetic)
M Mono    (optical)
```



existing one. For (a), this modification would change the value of the element but leave the element structure unchanged. For the other approach, no existing values would be changed. Assume we want to differentiate between changes of value and changes of structure. In this case, modeling aspects must be taken into account.

If we consider the data, we see that *Stereo* is the primary information while the rest of the line provides additional information. So, *West Rec* and *magnetic* are metadata for *Stereo*. In (d), this is modeled by additional sub elements with key *D* (descriptive). This structure is often contained in the original data of the collections, however, in the case of IMDB, it has to be extracted. This process requires domain knowledge and often has more than one solution. An expert, for example, might consider *magnetic* as a central aspect of the data as well. Hierarchical separation allows us to differentiate between modifications to primary and secondary data. It is also a perquisite for most value move edit types. Before we can compare this feature for different keys and/or collections, we must make sure that the degrees of hierarchical separation are similar enough to support this.

To generate data models that allow basic comparisons, we enforce the principles we discussed in Section 2.1.1: (1) Values of data elements are self-contained. (2) Data elements with the same key denote the same type of data. (3) Sub elements are not independent from their parents. The way these rules are applied can influence the outcome of our analysis. We add the point that, if possible, the number of data elements should be small. In particular, we use arrays in sub elements whenever possible as shown in (d).

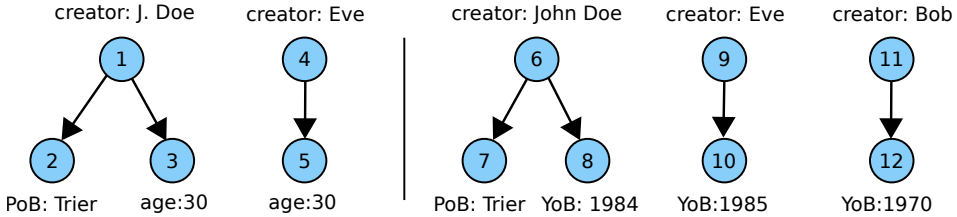
Chapter 3

Detecting and Categorizing Modifications

Contents

In the previous chapter, we discussed an observation-based model that provides a description of a record's history. In this model, the history consists of a chain of revisions. Each revision has a state that we informally defined as the sum of all data elements in the record. So far, we assumed that we could determine whether two states are equal or not. In this chapter, we will discuss the comparison of revisions in more detail, in particular, we present a framework that allows us to group related modifications and to categorize them. The ultimate goal of this work is to reconstruct the intention of the person (or algorithm) that caused the modifications. Consider the following example:

Example 3.1: Two revisions of a record. Keys and values are given, the numbers identify the data elements before mapping.



The left side shows the tree representation of the old state and the right side shows the current revision state. Obviously, there are several differences: The head element with value *J. Doe* disappears and two head elements with values *John Doe* and *Bob* appear. By looking at the example, we might come to the conclusion that three different modifications are mixed here. (1) The abbreviated name *J. Doe* was extended to *John Doe*, a very typical modification in many collections. (2) Another creator was added and (3) the data curator decided to change the metadata definition and transform the age element into a YoB (year of birth) element. These three modifications were observed at the same time, however, they are independent. Each modification has a different intent and to properly understand the history, we must consider them separately. There are several problems involved in this process. In the example, we assumed that *John Doe* is a modified version of *J. Doe* while Bob is an entirely new data element. However, the following scenario is also possible: (1) *J. Doe*'s real name is discovered to be *Bob* while (2) another (unrelated) creator with the name *John Doe* was added to the record. For the human observer, it is clear that the first scenario is more likely as *J. Doe* and *John Doe* are very similar and both head elements have a common sub element (PoB). However, we will see that this distinction is algorithmically challenging. The transition from *age* to *YoB* might be difficult to interpret even for a human observer. The key *YoB* is difficult to understand without further documentation and the values are also different. However, if similar modifications are observed for a number of different records, it might be easier to make sense of them.

In this work, we are dealing with large collections. To manage those, we need to approximate the human observers with a number of modification classification heuristics. To make this process manageable, we split the modification analysis into four different steps with different objectives. We first attempt to make sense of local observations. We then generalize these observations to the whole collection. Our framework is constructed in a way that the different steps can be adjusted to the collection that is currently studied. Figure 3.1 gives an overview of the framework. First, we will briefly present the steps. In the remainder of the chapter, we will discuss them in full detail.

Step 1: Alternations

In the first step, we compare the states of a record from two succeeding observations to determine if there are any modifications at all. If there are modifications, we want to determine the sets of added and removed data elements. Each of these data elements determines an alternation. For the alternation detection, we must consider the position of an element. Consider Example 3.1: we obtain six add alternations (data elements numbered 6,7,8,10,11 and 12) and four remove alternations (data elements 1,2,3 and 5). Data elements 4 and 9 are identical in position, key and value and are therefore not alternations. Data elements 2 and 7 are considered as removed and added, respectively, as they have a different parent element. The main purpose of step 1 is to filter data as preparation for further steps. The execution time of these steps scales with the number of alternations. Therefore, it is important to minimize the set of alternations.

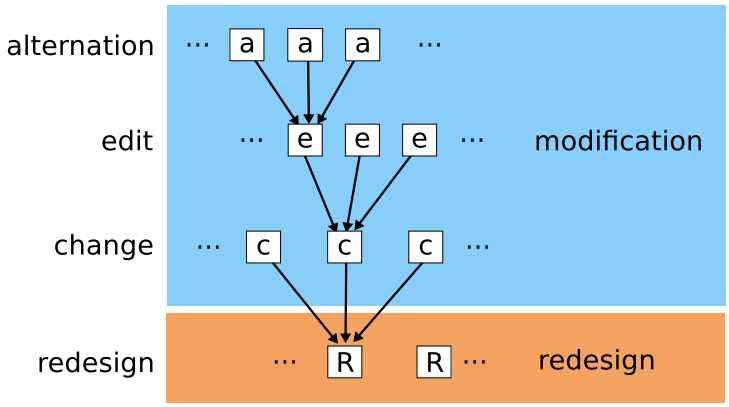


Figure 3.1: The 4 steps of the modification detection and categorization pipeline. a: alternation, e: edit, c: change, R: redesign.

Step 2: Edits

This step groups alternations from the same record that are related. In Example 3.1, we already discussed that data element 6 is a modified version of data element 1. The two alternations involved here (add data element 6, remove data element 1) are grouped into a *value-changed* edit. A single edit can cause an arbitrary number of alternations. If we determine that element 1 is the same element as element 6, then elements 2 and 7 have the same parent and are completely identical. I.e., the alternations we detected for those data elements are observation artifacts created by the modification of data element 1. These alternations must be considered when computing edits.

Step 3: Changes

Assume that there are several records that list *J. Doe* as creator. Further, assume that we detect several edits that rename *J. Doe* into *John Doe* in different records and that these edits occurred during a short period of time. These edits are isolated as they affect different records. However, if the observed edits occur at the same, or very similar time, we can assume that they are related by a common intent. A likely explanation would be that a data curator discovered the full name of *J. Doe* and fixed it throughout the collection. A change groups such edits with similar modification patterns and similar observation time.

Step 4: Redesigns

A redesign is a structural change to the data definition itself. For example, changing the *age* element in Example 3.1 to a *YoB* element throughout the collection would require a number of different edits/changes for the records. *Redesigns* group change operations and often represent updates to the underlying metadata model. The line between change and redesign is blurred. In general, redesigns group modifications that affect keys rather than values of data elements. They also occur over a longer span of time.

We compute these sets of modifications one after another. Note that it would have been possible to combine *alternation* and *edit*. We decided against this for two reasons. (1) The number of modified data elements is small compared to the total number of data elements for all our data sets. This is not surprising as complete replacements of metadata records are unlikely. We found it useful to determine the affected data elements by a simple procedure while the more complex edit detection is only applied to the modified sub-set. (2) The edit detection has to be tuned to specifics of a collection. Splitting alternations and edits allows us to test several settings for edit detection without repeating the time-consuming alternation extraction.

We will now discuss the four different steps of our pipeline in detail. Some steps have preconditions related to the data categorization we discussed in Section 2.3.

3.1 Detecting Alternations

In the first step, we compare consecutive states of the same record and identify data elements which have been added or removed. We call the adding and removing of a data element an alternation:

> ***Definition 3.1 (Alternation):*** Let $r\langle t_1\rangle$ and $r\langle t_2\rangle$ be two observed states with $t_1 <_\mathcal{T} t_2$. Let $e \in r\langle t_2\rangle \wedge e \notin r\langle t_1\rangle$. We call $a := (e, r, t_2)$ an **add alternation** of r. We denote $\mathcal{A}^+_{r,t}$ the set of all add alternations of r at time t.
>
> Similarly, let $e \notin r\langle t_2\rangle \wedge e \in r\langle t_1\rangle$. Then we call $a := (e, r, t_2)$ a **remove alternation** of r. $\mathcal{A}^-_{r,t}$ is defined similarly to $\mathcal{A}^+_{r,t}$.

Definition 3.1 is based on the definition of $e \in r\langle t_1\rangle$, which we have not given so far. As pointed out in Section 2.1.1, a unique identifier on data element level is not available for many data sets. To identify a data element in a set of data elements, we define a number of similarity measures. A data element is primarily characterized by its key and its value:

> ***Definition 3.2 (Local Identity):*** Let e_1 and e_2 be data elements with values v_1 and v_2 and keys k_1 and k_2 respectively. We call e_1 and e_2 **locally identical** if $v_1 = v_2 \wedge k_1 = k_2$. We write $e_1 \overset{L}{=} e_2$

If the value is an array, we demand pairwise identity of the array components. If a record has complex data elements, the local identity might be insufficient. Consider the following record where two creators are stored with their mail addresses as sub elements.

Example 3.2: Record with two creators

```
creator: [ Adam
  mail: bob@uni-trier.de ]
creator: [ Bob
  mail: adam@uni-trier.de ]
```

Apparently, the mail addresses were swapped. If a modification corrects this error, all data elements of the old revision will have a locally identical data element in the new revision. However, as the mail addresses are semantically bound to their parent elements, we want to track such modifications as well.

Definition 3.3 (Top Identity): For data elements e_1 and e_2, let p_1 be the parent of e_1 and p_2 be the parent of e_2. We call e_1 and e_2 **top identical** (denote $e_1 \overset{T}{=} e_2$) if $e_1 \overset{L}{=} e_2$ and $p_1 \overset{T}{=} p_2$ or $p_1 = p_2 = null$.

I.e., we demand that data elements e_1 and e_2 have the same top-path. The final similarity aspect of data elements is the similarity of their children.

Definition 3.4 (Children Identity): With e_1 and e_2 as above. Let s_1 and s_2 be the sets of sub elements of e_1 and e_2 respectively. We call e_1 and e_2 **child identical** (denote $e_1 \overset{C}{=} e_2$) if $e_1 \overset{L}{=} e_2$ and $\forall s \in s_1 \exists s' \in s_2 : s \overset{C}{=} s'$.

Locally identical simple data elements are also child identical. We differentiate between top and children identity as changes to the position of a data element are usually far more relevant to the information in a record than changes to the children.

The algorithm which we use to find alternations employs the different identities to create a matching between the data elements of different revisions. Data elements that cannot be mapped are either added or removed. Before we start, we transform the forest of data elements into a tree by adding an artificial root node. All head elements are children of this node. The task of comparing the records for modifications is thereby transformed into the task of comparing two trees. There is extensive previous work on this topic, especially in the field of comparing XML documents, which are also trees. Luuk Peters [Pet05] provides a good overview of the problem. However, if we look at common algorithms, we find that they often make assumptions on the trees that do not apply to our data. (1) The depth of the metadata tree is smaller than the average depth of most XML documents. Most data elements have only a single layer of sub elements. (2) The order of children is not important as data elements have no natural order. If order is needed, we use the order rule mechanism described in Definition 2.5. (3) In most XML documents, the actual

data is usually stored in the leafs while the ancestor nodes provide structure and categorization. I.e., two subtrees will most likely not be considered as identical if their leafs are different. In a metadata tree the situation is reversed. Here, the head elements contain the actual information while the sub elements contain information which describes this information (metadata for the metadata). I.e., the primary data can be untouched by a modification while some support information is altered.

In particular, we need an algorithm that takes the importance of high level data elements into account. Therefore, we define a matching between states based on the top similarity of an element.

Definition 3.5 (Top Matching): Let $r\langle t_1 \rangle$ and $r\langle t_2 \rangle$ be two states of record r. Let E_1 and E_2 be the set of all data elements (including sub elements) of $r\langle t_1 \rangle$ and $r\langle t_2 \rangle$ respectively.

Let $A \subseteq E_1$, $D \subseteq E_2$ and $M \subseteq E_1 \times E_2$. We call $\mathcal{M} := (A, D, M)$ a **top matching** of $r\langle t_1 \rangle$ and $r\langle t_2 \rangle$ if

1. $\forall (e_1, e_2) \in M : e_1 \overset{T}{=} e_2$
2. $E_1 = D \cup M(\cdot,)$ and $D \cap M(\cdot,) = \emptyset$
3. $E_2 = A \cup M(,\cdot)$ and $A \cap M(,\cdot) = \emptyset$

We then call $a \in A$ the **added elements**, $d \in D$ the **deleted elements** and $m \in M$ the **matched elements**.

Each element in A defines an add alternation. Each element in D defines a remove alternation. Assume the value of a data element with sub elements is modified. The old revision of the data element will be in D and the new revision will be in A. As the old and the new revisions are not locally identical, the sub elements cannot be part of M even if they are unchanged.

For two states there can be multiple top matchings. Obviously, a matching where all elements in $r\langle t_1 \rangle$ are assigned to D and all elements in $r\langle t_2 \rangle$ are assigned to A is a valid matching. However, as the primary function of the alternation extraction step is to reduce the data load of the further steps, this matching is useless. In general, we prefer matchings where A and D are small:

Definition 3.6 (Matching Quality): Let $\mathcal{M} = (A, D, M)$ be a matching. We define the **quality of the matching** as

$$q(\mathcal{M}) = \frac{2 \cdot |M|}{2 \cdot |M| + |A| + |D|}$$

If and only if the two trees are identical, then the matching quality is 1. If there are multiple matchings, we prefer the matching with the highest $q(\mathcal{M})$. Testing all

possible matches is infeasible for non-trivial records. To obtain a good matching in acceptable time, we apply a greedy heuristic based on blocking of key and values. We consider the following properties of metadata records:

- Parent elements are more relevant for comparison than child elements.
- Most metadata records contain a small amount of locally similar elements.

Algorithm 1 shows an outline of our approach:

Algorithm 1: findMatching

```
   input  : E1, E2, (A, D, M)
   output: (A, D, M)
 1 B1 ← block E1 by local identity;
 2 B2 ← block E2 by local identity;
 3 for b1 ∈ B1 do
 4     if ∃b2 ∈ B2 : b1 ≙ b2 then
 5         if |b1| = |b2| = 1 then
 6             M ← M ∪ (b1 × b2), B1 ← B1 \ b1, B2 ← B2 \ b2;
 7             (A, D, M) ← findMatching(children(b1), children(b2));
 8         else
 9             (A, D, M) ←matchChildren(b1, b2, (A, D, M);
10     else
11         D ← D ∪ b1 ∪ descendants(b1), B1 ← B1 \ b1;
12 for b2 ∈ B2 do
13     A ← A ∪ b2 ∪ descendants(b2), B2 ← B2 \ b2;
```

Algorithm 1 compares data element sets E_1 and E_2. Let H_1 and H_2 be the sets of head elements from $r\langle t_1\rangle$ and $r\langle t_2\rangle$ respectively. We call *findMatching*$(H_1, H_2, (\emptyset, \emptyset, \emptyset))$ for the head sets and an empty matching. In lines 1 and 2, we compute a blocking based on the local identity. We then iterate over the blocks in B_1 and search for identical blocks in B_2. Here, $b_1 \overset{L}{=} b_2$ means that all data elements in b_1 and b_2 are pairwise locally identical. If we do not find a matching b_2, we consider the data elements in b_1 and all of their children to be removed. A b_2 for which there is no partner b_1 contains added data elements. In case of a unique match we continue the algorithm with the child elements of the elements in the matched block. This order ensures that all matched elements are top similar.

If there are multiple candidates for a match (lines 8+9), we need to find the local matching which best matches the children of the data elements in b_1 and b_2. Algorithm 2 (matchChildren) computes all top matchings between pairs of blocked data elements. The matchings are ordered in a priority queue PQ so that the best match is first. The algorithm then greedily merges elements until only added and removed elements remain. *matchChildren* is expensive because of the quadratic

number of matches that are computed. However, the algorithm is practically usable as non-unique blocks are small and the sub element trees are shallow.

Algorithm 2: matchChildren

input : B_1, B_2, (A, D, M)
output: (A, D, M)

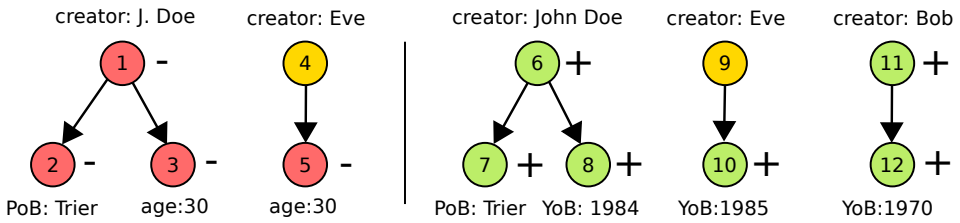
```
1  PQ ← ∅ ;
2  for f1 ∈ B1 do
3      for f2 ∈ B2 do
4          PQ.push(f1, f2, matchChildren(f1, f2));
5  while PQ not empty do
6      (f1, f2, (A', D', M')) ← PQ.pop();
7      (A, D, M) ← (A ∪ A', D ∪ D', M ∪ M');
8      remove all (f1, ·, ·) and (·, f2, ·) from PQ;
9      B1 ← B1 \ f1, B2 ← B2 \ f2;
10 for f ∈ B1 do
11     D ← D' ∪ f ∪ descendants(f);
12 for f ∈ B2 do
13     A ← A' ∪ f ∪ descendants(f);
```

In a first step (lines 2-4), the algorithm computes a matching of all pairs. Since f_1 and f_2 are already top identical, the only difference is how similar their children are. *matchChildren* is a variation of Algorithm 1, which returns a matching between the children of f_1 and f_2. The matchings are stored in a priority queue in descending order of matching quality. This step is feasible as most metadata trees are shallow.

In lines 5-9, the algorithm greedily selects the best matching from the priority queue PQ and adds it to the result. This may include cases where the *matchChildren* returned a matching quality of 0. Low matching qualities are not relevant here as the top identity is the most important factor. Leftover blocks are added to A and D (lines 10-13).

The example below shows the initial example annotated with alternations.

Example 3.3: Example 3.1 with annotated add and remove alternations.



3.2 Detecting Edits

Alternations represent data elements for which there is no top similar partner in the other state of an observation. However, many modifications to metadata not only add or remove data elements but modify existing ones. Consider again the initial Example 3.1 (Page 41). For the human observer, elements 1 and 6 are quite similar. The person name has been extended and one of the sub elements (elements 3 and 8) has been significantly modified. We could assume that elements 1 and 6 actually represent the same data – here, the same person – only with a slightly different *surface form*. During the alternation detection, we extracted a remove alternation for element 2 and an add alternation for element 7. This is because the data elements are locally identical but not top identical. If we consider elements 1 and 6 as the same element, elements 2 and 7 can be matched and are no longer alternations.

We call the situation where the properties of a data element are modified but the old and the new properties are similar an edit.

Definition 3.7 (Edit): Let A^+ and A^- be the add and remove alternations observed for record r at time t. For alternation x let C_x be the set of alternation which affected the descendants of the data element affected by x. Let $C'_x \subseteq C_x$.

For $a \in A^-$ and $b \in A^+$ let $a \approx b$ and $\forall a' \in C'_a \; \exists b' \in C'_b : a' \overset{L}{=} b'$ and vice versa. We call

$$e := (a, b, r, t, C'_a, C'_b)$$

an **(modification) edit**. We call C'_a and C'_b **obsolete alternations**.

For $a \in A^- : \nexists b \in A^+ : a \approx b$ we call

$$e := (a, null, r, t, C_a, null)$$

an **(remove) edit**. Similarly, we call $e := (null, b, r, t, null, C_b)$ an **(add) edit**.

We write $key_{old}(e)$ and $value_{old}(e)$ to refer to key and value of the data element affected by alternation a. If $a = null$, we write $key_{old}(e) = null$ and $value_{old}(e) = null$. In the same way we write $key_{new}(e)$ and $value_{new}(e)$ to refer to key and value of the data element of alternation b. Note that *add* and *remove* edits are defined with all descendant alternations. Again, this is a result of the importance of high-level elements in the metadata tree. If an alternation has been classified as *add*, all its descendants will be classified this way too. The same is true for *remove* edits.

The central concept of edits is that the elements which are added and removed by the alternations are similar but not identical. This is expressed by $a \approx b$. We will discuss the similarity of data elements on the following pages. In Example 3.1, we obtain an edit $(1, 6, r, t, \{2\}, \{7\})$. The edit does not affect elements 3 and 8 as they are not similar even if we assume that elements 1 and 6 are identical. However, if we

conclude that the values are similar (which is not straightforward), we might create another edit $(3, 8, r, t, \emptyset, \emptyset)$.

We can categorize edits by considering the added and removed alternations:

add : An edit which adds a data element (including its sub elements). No similar data element has been removed in the same observation.

remove : An edit which removes a data element (including its sub elements). No similar data element has been added in the same observation.

modifyValue : The value of a data element is modified. Values are arrays of atomic information. We consider the following subtypes: **addValue**, **removeValue**, **replaceValue** and **reorderValue**. Reorder can only be applied if the array components have a natural order.

modifyKey : The key of the data element is changed.

reorderElement : Data elements do not occur in the same order as in the previous observation.

An edit can have more than one type (e.g., modifyValue and modifyKey) although detecting such edits is difficult as the old and new properties of the element would be quite different. We also considered an edit type where data elements are moved in the metadata forest. However, the fact that child elements are less self-contained than their parents makes this operation unlikely. For DBLP and IMDB, we considered data elements which are reassigned to another parent element. In each case, key and value where nondescript (96% were one digit numbers) and it was not clear if the curator actually intended a move or if the observation was a coincidence. For this work, we will ignore move edits.

When computing all edits for a record, we want to cover all alternations we observed.

> ***Definition 3.8 (Edit Set):*** Let A^+ and A^- be the add and remove alternations observed for record r at time t. Let $E_{r,t} = e_1, \ldots, e_k$ be a set of edits obtained for this observation. For $1 \leq i \leq k$ let $e_i := (a_i, b_i, r, t, A_i, B_i)$.
>
> We call $E_{r,t}$ an **edit set** if $\bigcup_{1 \leq i \leq k} (a_i \cup A_i) = A^-$ and $\bigcup_{1 \leq i \leq k} (b_i \cup B_i) = A^+$.

An edit set is a partition of the alternations from a specific observation of a specific record.

In the remainder of this section, we discuss how to extract edit sets and classify edits. Central for this is the value of the data element. We will see that value modifications are far more common than modifications to the key. This is not surprising as keys are strictly defined in the metadata definitions, which are quite stable. A value on

the other hand comes from a – potentially large – value domain and there are far more reasons why a value would change. In the following section, we discuss the problem of comparing values of data elements. In Section 3.2.2, we describe an algorithm for edit set detection.

3.2.1 Comparing Values from Metadata

There are four aspects that describe a data element, its ancestors, its children, the key and the value. As most metadata forests are shallow, children and ancestors are often not relevant. The key is a quite stable property that rarely changes. Thus, most modifications to data elements affect the value. To determine the similarity of modified data elements, we need a reliable measure for the similarity of element values. Consider the following example of two revisions:

> ***Example 3.4:*** Revisions of record.
>
> Revision 1: `author: A. Müller, author: B. Wagner`
>
> Revision 2: `author: Adam Müller, author: Bert Wagner`

Obviously, all author elements were modified. If we consider `A. Müller` and the elements in revision 2, there are three possible explanations:

a) `A. Müller` and `Adam Müller` represent the same data element with a changed value.

b) `A. Müller` and `Bert Wagner` represent the same data element.

c) `A. Müller` is not related to the author elements in the second revision. This element was removed and replaced by some other information.

To the human observer, it is obvious that solution a) is by far the most likely explanation. Both values are very similar. If the observer has some experience with handling names in libraries, she will know that extending abbreviated first names is a common operation which aims at improving the data quality. However, determining this algorithmically is difficult. In general, metadata values can be arbitrary strings. Comparing strings is a relevant problem in many fields and there are many different algorithms to compute string similarities. For a comparison of the more basic approaches, see [CRF03]. We define:

> ***Definition 3.9 (String Distance Function):*** A function
>
> $$d : \Sigma^* \times \Sigma^* \mapsto [0, 1]$$
>
> is called a **string distance function** if $\forall a, b \in \Sigma^* : a = b \Rightarrow d(a, b) = 0$.

Note that we do not demand $d(a, b) = 0 \Rightarrow a = b$. Using a single string distance function is insufficient. To illustrate this point, consider the following simple but frequently used distance functions:

The (normalized) Levenshtein distance [Lev66] measures the minimal number of operations which are necessary to transform a given string into another. In the most simple form, permitted operations are *add letter*, *remove letter* and *replace letter*. The Levenshtein distance between *Muller* and *Müller* is 1 as one letter has to be replaced. The normalized Levenshtein algorithm is scaled by the length of the longest input word to obtain a function as described above. Let s be the scaled Levenshtein value. To suit Definition 3.9 we use $1 - s$ as distance value.

The Jaccard distance considers strings as sets of tokens. A token is the result of a split operation performed in the input. For example, we might split the input on whitespace. For token sets A and B, the distance is defined as $1 - \frac{|A \cap B|}{|A \cup B|}$. The distance of `Martin Müller` and `Martin A. Müller` is $1/3$. Instead of words, smaller units such as n-grams or larger units such as terms can be used.

The quality of the results of Levenshtein and Jaccard depends on the input. Consider the following example:

Example 3.5:

p_1 Mratin Müller – Martin Müller

p_2 Wei Wang – Wang Wei

p_3 1999 – 2000

p_4 dblp.hostname.de – informatik.hostname.de

	p_1	p_2	p_3	p_4
Levenshtein	0.154	0.75	1.0	≈1.0
Jaccard	0.667	0.0	1.0	1.0

p_1 and p_2 represent typical problems related to personal names. For p_1, Levenshtein has the best performance. Had we used a common extension of the algorithm where swapping letters is also a base operation (Damerau–Levenshtein distance), the result would have even been better. For p_2, the Jaccard distance identifies both strings as identical as order is not relevant here. Levenshtein considers the strings as quite dissimilar.

> ***Observation 1:*** A single string distance function is insufficient to compare general pairs of values.

To obtain sufficient results, we need to combine multiple string distance functions. Reuther [Reu07] proposes an approach with a fixed set of functions. The distance of a pair of values is computed as the weighted average of these functions. In this work, we follow a different approach. We already saw that some distance functions are particularly well suited to detect specific kinds of modifications. Assume, we consider distance functions A, B and C, where each function detects a specific type of modification. A pair of values is a modification if one of the functions returns a small distance, regardless of the result of the other functions. I.e., to determine if a value pair should be matched, we consider the best result:

> ***Definition 3.10 (Atomic String distance):*** Let $d_1, ..., d_k$ be string distance functions and s_1 and s_2 strings. We call
>
> $$dist(s_1, s_2) := \min_{0 \leq i \leq k} d_i(s_1, s_2)$$
>
> the **atomic string distance**.

Definition 3.10 avoids the problem of finding appropriate weights for the different distance functions, which is a central problem in the approach of Reuther. We are also able to detect types of relations between values which are only covered by a single distance function.

p_3 is considered very dissimilar by both algorithms as there are no common letters or words. However, if we assume that the values of p_3 represent years, than the example shows the minimal distance between two non-similar values. Even worse, for a pair *4000* and *2000*, Levenshtein would have found a smaller distance. I.e., the year 4000 would have been more similar to the year 2000 than the year 1999. p_4 represents a similar constellation with URLs. In this case, the fact that the host (*hostname.de*) is identical is far more important than differences in the sub-host (*dblp* vs. *informatik*).

> ***Observation 2:*** Specific data types require domain-specific string distance functions.

All values we consider in this work are stings. However, the metadata definition might impose a certain format for values of a specific key. These definitions can cause the domains of two keys to be quite different. E.g., it might demand that the values of a specific key are dates expressed in a certain format. The domain of

another key contains only personal names. A simple idea to handle specific value domains is to select string similarity functions that are good at dealing with the expected values. I.e., for a value domain with numbers, we would select a similarity function which parses the string which represents the number. The comparison can then use the parsed numbers. There are three problems with this approach:

- There are many different value types to consider. It is also important to take the encoding and formatting conventions of a digital library into account. E.g., there are many different ways to transform a date into a string.
- The *value type* of a domain is tentative and often not strictly enforced. As discussed before, this is to allow the adding of unexpected values.
- Some of the digital libraries we observed provide several hundred different keys that each require a set of distance functions.

Alternatively, we can determine the set of distance functions automatically based on properties of the value domain. In his master's thesis, Thomas Kirsch [Kir14] considered the problem of classifying the datatype of a value domain. Kirsch defined the following data types: *named entity* (including personal names, organization names and place names), *number* (including integer and float), *date*, *url*, *text* (including short texts (titles etc.) and long text) as well as a category for strings defined by common *standards* such as ISBN. Kirsch evaluated different supervised learning algorithms to determine if they can reliably classify these data types of value domains. The classifiers were trained and evaluated with data from our test collections. Kirsch found that the accuracy of this approach is about 80%. However, it turned out that some data types are difficult to differentiate. Kirsch considered a further distinction of *named entity* into *personal name*, *organization name* and so on. These classes proved to be more difficult to classify and introduced a number of misclassifications. For this work, we added another value domain category, the *enumeration*. An enumeration domain contains values which are either identical or not identical with nothing in between. I.e., the values 100 and 101 are as similar as 1 and 999. These domains often describe categorical information. We define enumerations manually as they are difficult to distinguish from other categories. For each value class, we define a number of string distance functions which work well with these data. For example, we use a string distance which parses common date formats and compares the actual dates instead of the strings. Each string distance function evaluates if it can be applied for a pair of values. This is necessary to handle unexpected values. The functions which agree to handle a pair of values are used to compute the atomic string distance. In case that there is no function which will work with a data set, we assume maximum distance of the values.

In Appendix B.1, we briefly describe the similarity functions we used for this work. The string similarity functions we discuss there are static, i.e., they rely on a fixed algorithm and possibly a small data set like a list of stopwords. Alternatively, we could use learning-based approaches that can adapt to the properties of the name pairs.

Bilenko et al. [BMC+03], considered the problem of comparing names of persons, places, organizations and similar data. However, most learning-based approaches require a teacher. In most cases, the teacher is a data set from which the algorithm can learn relevant properties of the data. Some approaches are self-learning, i.e., they require no explicit learning data. Regardless, they need adjustment to the current problem. As we analyze a large variety of data sets with many – often very different – value domains, we decided to use the simpler and more robust approach explained above. The base algorithms we use are also very fast, which is important as we handle large data sets and need to compare many value pairs.

So far, we only discussed atomic value domains and ignored domains which consist of arrays. In the data sets we consider in this work, atomic domains are dominant. However, we need to handle array type values as well. To improve the readability of this section, we moved the discussion of value similarity for values to Appendix B.2. In the appendix, we describe how the atomic value similarity can be extended to compare arrays of values.

3.2.2 Computing Edits

We now use the value similarity to compute edit sets as described above. At the end of the computation, each alternation will be assigned to exactly one edit.

Let T_1 be the metadata tree of a record before an observation and T_2 the tree of the same record after an observation. As before, we assume that the head elements of the underlying records are children of an artificial root node so that the metadata actually form a tree. Let both trees be annotated with alternations of that observation: A^+ and A^-. We define a tree that is merged from T_1 and T_2.

> ***Definition 3.11 (Joint Alternation Tree):*** We call T a **Joint Alternation Tree** of T_1 and T_2 if
>
> - it contains all data element which are affected by alternations from A^+ or A^-.
> - for element f of an alternation in A^-, T contains all ancestors of f in T_1.
> - for element f of an alternation in A^+, T contains all ancestors of f in T_2.
> - it contains no other data elements.

To create T, we iterate over the set of alternations and add nodes if needed. If we store the current content of a tree in a hash set, we can determine if the parent is present in O(1). As an alternation is processed at most once and the computation stops if a parent is found, we process each edge in the basic trees at most once. We assume that the artificial root of both metadata trees match so that T is connected. This way, the artificial node becomes the root of T. Figure 3.2 shows a small example.

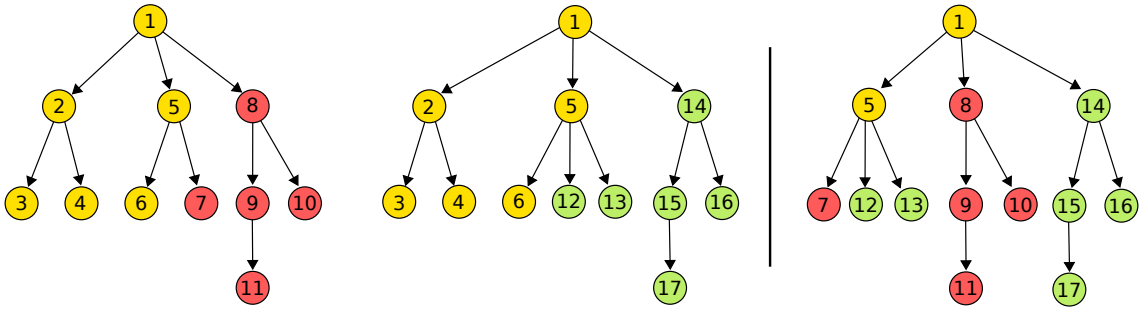


Figure 3.2: Two metadata trees and their joint alternation tree (right). Green nodes are added, red nodes are removed and yellow nodes can be found in both revisions. Nodes 2, 3, 4 and 6 do not appear in the joint alternation tree as they are not affected by alternations and not ancestor of a data element affected by alternations. Node 1 is the artificial tree root.

We traverse the joint alternation tree from top to bottom. For each node n we determine n^+: the number of children of n which are affected by an add alternation. In the same way, we determine n^- for remove alternations. Similar to the detection of alternations, we pairwise compare the nodes in n^+ and n^- to find the best match. The differences to the comparison for alternation detection are:

- We need to process fewer nodes, as alternation detection has already removed matching data elements from consideration.
- We now permit deviating keys and values. E.g., we forgo the blocking step of Algorithm 1 (Page 47).

To compare two data elements, we compute the edit element distance:

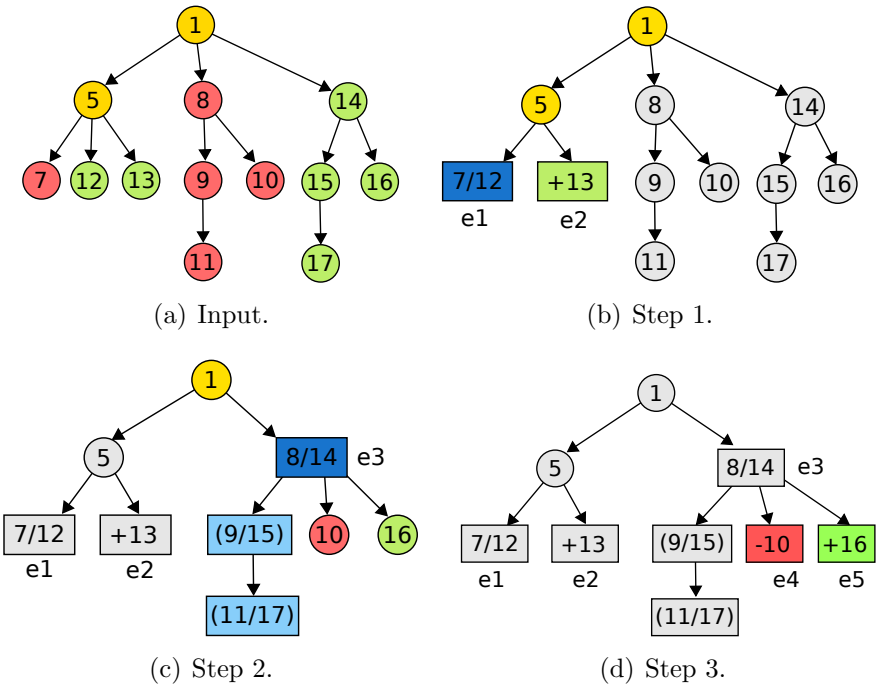
Definition 3.12 (Edit Element Distance): Let n be a node and $a \in n^+$ and $b \in n^-$.

We define the **edit element distance** as

$$dist_{\text{edit}}(a, b) := dist_{\text{value}}(a, b) + dist_{\text{children}}(a, b) + dist_{\text{key}}(a, b)$$

where $dist_{\text{value}}(a, b)$ is the value distance as described above and $dist_{\text{children}}(a, b)$ is the quality of a matching of the descendants of a and b (see Definition 3.6). We define the key distance as

$$dist_{\text{key}}(a, b) := \begin{cases} 0 & \text{if } key(a) = key(b) \\ 1 & \text{if } key(a) \neq key(b) \end{cases}$$



(a) Input. (b) Step 1.

(c) Step 2. (d) Step 3.

Figure 3.3: Extraction of edits from the joint alternation tree in Figure 3.2. A total of five edits are extracted. (9/15) and (11/17) represent pairs of obsolete alternation.

We considered using a key similarity which returns any value in $[0, 1]$ based on a similarity measure between the keys themselves. However, we were not able to extract key similarity from the data collections we consider in this work.

If the edit distance is below a certain value δ_{edit}, we merge the two nodes into a joint tree node which then represents the edit. We use the child matching we computed to determine obsolete alternations. To determine δ_{edit}, we selected 100 records from each collection and manually annotated the edits. For Wikipedia, we selected records for the largest template set of each language: a total of 300 records. For all collections, $dist_{\text{value}}$ is the most important part of the comparison as (1) key changes are rare and (2) many data elements do not have children. In addition to threshold δ_{edit}, we also need to set a maximum value distance δ_{value}. This is to prevent the matching of data elements which have the same key, possibly no children and completely different values. Through experiments with the annotated records, we determined values $\delta_{\text{value}} = 0.3$ and $\delta_{\text{edit}} = 1.1$. This means that key changes are detected as edits only if values and children are quite similar. $\delta_{\text{value}} = 0.3$ and $\delta_{\text{edit}} = 1.1$ produce good results for all collections we considered. However, we only annotated a very small amount of records and the results might be misleading.

If nodes a or b have children which are not marked as obsolete, the algorithm continues to process them. Nodes in n^+ and n^- which remain unmatched are considered to be add and remove edits respectively. Figure 3.3 shows the algorithm working on the example from Figure 3.2. Figure 3.3(a) shows the input tree. In Figure 3.3(b), nodes

Table 3.1: Edits detected by project and the share of edits with a specific type. An edit can have more than one type. E01, D01 and F01 from Wikipedia, see Appendix A.2

project	edits	add	remove	value	key
DBLP	2,131,915	54.46%	14.36%	29.34%	0.018%
IMDB	35,276,078	69.89%	17.97%	11.89%	0.002%
CiteSeer	79,775,653	47.39%	26.43%	26.15%	≈0%
Open Library	124,920,902	66.91%	15.97%	15.80%	0.014%
E01	1,088,185	44.46%	44.40%	10.84%	0.004%
D01	1,834,049	22.67%	21.46%	55.44%	0.005%
F01	284,881	24.40%	10.32%	64.26%	≈0%

7 and 12 have been grouped into an edit while node 13 did not find a partner and was categorized as an add type edit. In Figure 3.3(c), nodes 8 and 14 are grouped into edit $e3$. The children mapping used in this step identifies node 9 as identical to node 15 and node 11 as identical to node 17. The alternations that define these nodes were extracted because their data elements were not top identical. However, we now consider the data element behind nodes 8 and 14 to be the same (though with some modifications), then the alternations are obsolete now. In the final step (Figure 3.3(d)) nodes 10 and 16 were not considered to be a match and processed to add and remove edits respectively.

We applied the edit detection to the test collections described in Section 2.4. Table 3.1 lists the number of edits and the share of edits with a specific type. Note that an edit can have more than one type though this is uncommon. For Wikipedia, we print the number of edits for the largest collections of each language. In all projects, key change edits are rare with $\approx 0\%$ – 0.18% of all edits being key change edits. Value changes are far more common with shares between 10.84% and 64.26%. We were surprised by the differences for the Wikipedia data sets given the fact that all three data sets contain the same type of data (biographic data). There are two possible explanations. (1) The performance of the edit detector varies for different collections. (2) The Wikipedia projects handle the underlying templates differently. The solution is unclear at the moment.

3.3 Detecting Changes

Alternations and edits are extracted on record level. I.e., for each edit there is exactly one affected record. However, some modifications can affect multiple records at once. Consider the following example with two records $r1$ and $r2$:

Example 3.6:

```
r1 : author = [A. Adam], author = [B. Bob], title = t1
r2 : author = [A. Adam], title = t2
```

Both records list author *A. Adam*. Assume that a data curator determined that the first name of this author is *Andrew*. Some collections – like DBLP or IMDB – try to store the full author name in each record. In the example, this means that the value of two author author need to be changed from *A. Adam* to *Andrew Adam*. As this affects two different records, we register two edits. These edits are related by their common cause – the curator wanting to complete the name of the author. Edits which are related in this way have two properties:

- The modifications made by the edits are identical or very similar.
- The edits occur in a limited time frame.

We call these sets of edits **changes**. The time factor is important as we attempt to group edits which are caused by the same intent. Two similar edits which occurred with a whole year in between them are most likely not caused by the same intent and do not belong to the same change. To extract changes, we need to determine the temporal proximity of edits.

Precondition: For the computation of changes:

- A global clock scope (see $\mathcal{T}_{\text{scope}}$ in Section 2.3) is required.
- A real time clock ($\mathcal{T}_{\text{clock}}$ in Section 2.3) is required.

The global clock scope is required, because we need to determine the temporal distance of edits. Changes can be computed with a logical clock. However, we need to know a reliable upper bound for the real time elapsed between two ticks of the clock. Of the collections which we consider in this work, only Citeseer does not meet the preconditions (logical and local clock).

In this work, we will only consider changes defined on edits with identical modifications.

Definition 3.13 (Modification Identity): Let e_1 and e_2 be edits. We consider e_1 and e_2 to be **modification identical** if $key_{old}(e_1) = key_{old}(e_2)$ and $key_{new}(e_1) = key_{new}(e_2)$ and $value_{old}(e_1) = value_{old}(e_2)$ and $value_{new}(e_1) = value_{new}(e_2)$.

We write $e_1 \overset{m}{=} e_2$.

Temporal proximity is more difficult to define. Whether two edits are close to each other depends on the edit policies of the collection as well as the temporal properties. For example, for IMDB, we observe the state of the collection once per week. In this case, it is reasonable to assume that edits of a change need the same timestamp to be temporally close. If a collection has a finer observation time line, we must consider cases where a change consists of edits from different timestamps. This is particularly true for collections with an event-based artifact creation trigger. To group edits based on temporal proximity, we use a simple sliding window approach:

Algorithm 3: findChanges

```
   input  : temporarily sorted edits E
   output: set of changes C
1  C ← ∅, C_open ← ∅ ;
2  for e ∈ E do
3      if ∃c ∈ C_open : e =^m c then
4          if t(e) − t_start(c) < τ ∧ t(e) − t_end(c) < ϵ then
5              c ← c ∪ e
6          else
7              C ← C ∪ c, C_open ← C_open \ c ;
8              C_open ← C_open ∪ {e};
9      else
10         C_open ← C_open ∪ {e};
11 C ← C ∪ C_open;
```

We iterate over all edits in temporal order. An edit can either be added to an open change (line 5) or added to a new change (line 8). An edit can be added to an open change c if it is modification identical to it (i.e., modification identical to all edits in c). Let $t_{start}(c)$ be the oldest timestamp of an edit in c and let $t_{end}(c)$ be the youngest timestamp. We demand that the new edit is not more than ϵ younger than $t_{end}(c)$ and that the total time interval covered by the change is no larger than δ. ϵ ensures that the edits in c are grouped without a significant temporal gap. τ prevents the time interval of a change from becoming too long. This is relevant for changes on edits which occur frequently but are not directly related. We base our definition of a change on Algorithm 3.

Definition 3.14 (Change): Let $E := E_1, \ldots, E_k$ be the partition of all edits generated by Algorithm 3.

For E_i with $1 \leq i \leq k$ and $e \in E_i$ a random edit in E_i and $t_{start} := \min_{e \in E_i} t(e)$ and $t_{end} := \max_{e \in E_i} t(e)$, we call

$$c := (E_i, t_{start}, t_{end}, key_{old}(e), key_{new}(e), value_{old}(e), value_{new}(e))$$

a **change** modification.

Table 3.2: Change count by project. changes: total number of changes, avg.: mean number of edits per change, max: highest edit count per change, edit count > 1: changes with more than one edit.

project	τ	ϵ	avg.	changes	max	edit count > 1
DBLP	2d	1d	1.427	1,441,728	33,233	72,124
IMDB	0	0	2.680	13,161,800	379,438	2,429,874
Open Library	12h	4h	2.611	47,836,764	13,745,406	3,247,850

By definition, all edits of a change have the same edit type. The type of a change is determined by the type of edits it is composed of. E.g., we refer to a change composed of add type edits as add change.

Table 3.2 lists the results of the change detection for the three major projects that meet the preconditions. The choice of τ and ϵ largely depended in the observation time frame of the projects. As pointed out above, IMDB is limited to weekly observations so changes with multiple observation dates make no sense. For DBLP, we observed many changes which are stretched over two days. This is caused by partial modifications which are finished after a checking algorithm ran overnight. Open Library provides event-triggered artifacts, i.e., the observation occurs at the moment of the change. We tested several settings for a random sample of the collection. We found that extending τ and ϵ beyond 12 hours and 4 hours respectively does not increase the size of changes significantly.

The average number of edits per change ranges from 1.427 to 2.680 with most changes consisting of a single edit. For all three collection we detected some changes with a very high number of edits. These changes represent technical changes to the collection metadata. For example, one change to Open Library consists of 13,745,406 edits. This change adds an empty tagging element to many records.

3.4 Detecting Redesigns

The final layer of the modification framework is the **redesign**. A redesign combines multiple changes into a modification which alters the structure of many metadata records simultaneously. Detecting a redesign algorithmically is difficult as it requires the grouping of changes based on a common intent. Possible redesign scenarios are:

Key Rename A key is consistently renamed in the collection.

Key Split Data which has been assigned to a single key before it is partially reassigned to a new key. Example: a new key is created to accommodate data which was previously stored with another key (together with other data). E.g., IMDB provides a data element for miscellaneous roles. Sometimes, new keys are created for more specific roles. We find several dates on which 100+ changes change the key from miscellaneous to a more specific description.

Element Split/Merge The value of a data element is split and now stored in different data elements. E.g., the author name was stored in a single data element but is now stored as *first name* and *last name*. An element merge is the reverse operation of an element split. Detecting this kind of redesign would require better or different value comparison algorithms in the edit detection stage. It would also require an extension of the edit framework so that, for example, a remove alternation could be mapped to multiple add alternations.

Value Transformation The values of specific data elements change in a systematic way. E.g., all ISBN-10 numbers are transferred to ISBN-13 numbers. We encountered a massive value transformation in IMDB when all personal names were changed from *firstname lastname* to *lastname, firstname* and then back again. At that time, we used a preprocessing to filter out these modifications as we needed the personal names as stable identifiers.

Redesigns can be extracted from edits and from changes. However, for edits we need to know temporal proximity of modifications so a global clock is required. Redesigns can be used to understand large-scale modifications to the metadata of the collection. However, detecting redesigns is challenging for a number of reasons.

- Redesigns might be spread over a longer period of time than changes. This makes intersection with *regular* changes or other redesigns more likely.
- The common properties of a redesign can be more subtle than for changes. I.e., detecting a split of a person name in first and last name elements requires a basic understanding on the structure of names.

In the remainder of this work, we will ignore redesign modifications. The extraction and understanding of this modification type should be addressed in future work.

Chapter 4

Understanding Defects

Contents

In this chapter, we discuss how historical metadata can help us dealing with defective records. A record is defective if it contains one or more wrong or misleading data elements. These data elements can interfere with the user experience of a digital library. Assume that a record for a book written by *John Adams* stores a data element *creator: John Aadms*. If a user searches for documents by *John Adams*, the book represented by the defective record might not be found. Another risk is

that the library, ignorant of the defective record, will create another record for the book with the correct metadata. Most libraries invest significant effort in preventing errors and removing existing defects. This includes the manual revision of the data (sometimes with the help of the library's patrons) as well as algorithmic solutions (e.g. see the two surveys [EIV07] and [FGL12] on technical approaches and [Ell10] which focuses on strategies of manual solutions).

In this chapter, we discuss how historical metadata can be used to improve the understanding of defective data. We concentrate on two questions: (1) what are the properties of defective entities, in particular, in comparison to non-defective data and (2) how can the efficiency of defect detection/correction efforts be evaluated. To answer the first question, we need to identify and study defects in a specific collection. For the second question, we need representative test collections on which we can apply the approaches. For these test collections, we need to know the so-called *ground truth*, i.e., we need to know which parts of the test collection are defective and how they should be corrected. Both questions are related. A good understanding of defect properties can help us design better ways to deal with them. A good test collection itself might be able tell us a lot about the typical defects of a data set.

The test collections which are currently in use are quite small. For example, Han et al. [HZG05] created a test set from DBLP which consists of about 8500 signatures. A signature is a single data element of a record which holds the name of an author. At the time of creation, this test set incorporated about 0.8% of DBLP. In 2015, the test set represents only about 0.15% of the collection. Levin and Heuser [LH10] gathered three data sets each consisting of a few hundred records. Ferreira et al. refer to these sets as *very small* [FGL12]. These collections are created by manually cleaning a small part of the underlying data set. This expensive step accounts for the size of the collection. There is also no guarantee on the correctness. For example, Shin et al. [SKCK14] analyzed the data set provided by Han et al. [HZG05]. They claim that 3.37% of the signatures in the ground truth are wrong. In addition to that, they were not able to verify information for another 22.85% of the signatures.

In this chapter, we propose a method to extract a significant number of defects from a collection based on the historical metadata. Assume that in our initial example, the library detected the misspelled name. The logical next step would be to correct the record. These corrections can be traced with our modification detection framework. We consider corrections as test cases. We assume that the data was defective before the modification and that the data is correct afterwards. Our approach has a number of advantages which complement the existing test collections:

- It uses implicit information extracted from the historical metadata. I.e., it can be used on all collections that provide sufficient historical data. It is not limited to sub sets of collections.
- In a realistically large collection, defects can be found in many ways. Our approach includes all defect corrections, regardless of what triggered them.

- We can identify corrections that were based on data that is no longer available. Shin et. al [SKCK14] were not able to verify 22.85% of the citations in the nine years old Han collection. However, Han claim manual checking of all citations, and it is very well possible that they were able to use data sources that are no longer available (e.g., deleted web pages, inactive/deceased researchers...).
- Within the constraints of the temporal data, we can track the development of a defect from its creation to the correction. This can give us insights into the origin of defects.

There are two central problems which we have to overcome in order to use historical data for defect analysis. First, we need to separate defect corrections from other modifications. Assume that a record stores the physical location of a book. If the book is re-shelved, the record has to change. However, neither the old location value nor the new location value are defective in the sense described above. To solve this problem, we first need to discuss the types of defects which occur in bibliographic metadata (see Section 4.1). The detection of defect corrections itself is an extension of the historical framework which we presented in Chapter 3. We discuss this extension in Section 4.2.1. The second problem is that our approach is biased by the defect detection strategy of the collection which we process and by the quality of our modification extraction framework. We discuss this problem in Section 4.2.2. In Section 4.3, we describe two test collections based on detected corrections and discuss scenarios for their application. In Sections 4.4 and 4.5, we discuss properties of defects that are related to names and network information respectively. We conclude this Chapter with a short outlook on how detected defects can be used to predict undetected ones (Section 4.6).

4.1 Defects in a Digital Library

All non-trivial data collections contain records that are defective in some way. Borisov et al. [BBMU11] identify hardware defects, bugs in software and mistakes by humans as the major causes for these defects. Based on our findings, which we describe in the following sections, we might add that insufficient and conflicting data sources are another major cause of problems. But what makes a record *defective*? A comprehensive view on data defects is provided by Thomas Redman [Red96]. Though Redman concentrates on business information processes, his work is applicable to digital libraries without modification. Reuther [Reu07] has shown that from the point of view of digital libraries, this model is similar to many other popular models.

Redman differentiates between three major views on data quality: the *conceptual view*, the *data values* and the *data representation*. For each view, Redman defines a number of quality dimensions. The conceptual view describes how well-defined the data model is. This view is central to Redman's work. In his opinion, defect prevention should focus on the design of data handling policies. The data representation

view deals with aspects of actual data storage and retrieval systems. Both views are of little importance here as neither the conceptual metadata definition nor the data representation are sufficiently represented in the historical metadata. In most cases, we do not have information on model and storage system. The only information directly available to us are the data values. For this view, Redman presents four dimensions of data quality:

Accuracy describes if the data quality is sufficient for an application. The central application for bibliographic metadata is document retrieval. For this task some data elements are very relevant, e.g., the creator element. We have high expectations in the correctness of these elements. Other information, e.g., the pagination data, might be considered accurate even if they are not exactly correct.

Completeness: a record without data elements is accurate but not useful. A metadata record is considered complete if all data expected by the users are present. In this work, we ignore expectancies on the presence of records themselves.

Currency: like all data, metadata can age. While many information like title and author names are quite stable, some data elements can become outdated over time. This is mostly true for administrative metadata such as the storage location of a book. Again, the expectations of the users are the relevant measure to determine when data are outdated.

Consistency: in the framework of Redman, data is consistent if it is conform to an authority. For example, the consistency of an ISBN can be checked mathematically. Consistency does not imply accuracy, the mathematically consistent ISBN, for example, does not need to be the correct one.

The quality dimensions are not orthogonal. Consider the following situation: a metadata record stores the physical location of a book (e.g. the location of the bookshelf in the library). If the library changes the storage system, the location information becomes outdated. At the same time, the record cannot be used to locate the book. Therefore, the data is also inaccurate. We could also argue that an important information – the location – is not present in the data set. This would make it also incomplete. This can be confused further by unexpected usage. Assume that a user looks up the location of the book just before the reorganization and plans to fetch it a few days later. At the moment of the lookup, the data is not outdated. However, the metadata is insufficient to serve the user. The last example shows that data quality must be considered in the context of *realistic* applications and their expectations.

In this work we concentrate on situations where a data element holds a value that is not correct. Eventually, the defective value might be replaced. This operation can be tracked with the historical metadata framework. We ignore currency-related defects

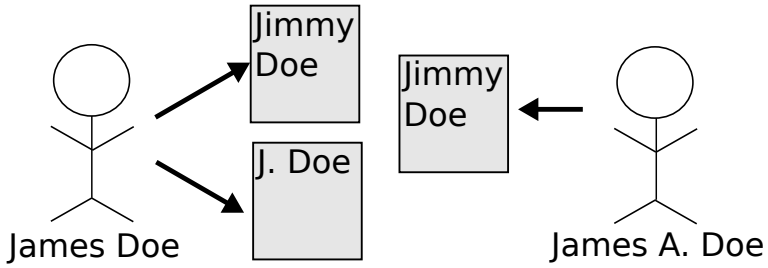


Figure 4.1: Example of person-name inconsistencies. The boxes represent publications, the stick figures represent person entities. *Jimmy Doe* is homonym for *James Doe* and *James A. Doe*. *J. Doe* and *Jimmy Doe* are synonyms for *James Doe*.

as it is difficult to tell when a data element becomes obsolete without considering the context of the collection. We also do not consider completeness on record level. Whether a record is in a collection or not is often related to the libraries policy and curation process. We will discuss general aspects of tracking record adding in Section 5.1.

Many defects in metadata records are local. For example, wrong pagination data in a record might make the publication more difficult to find if the user searches for publications with a certain number of pages. To correct this defect, only a single metadata record must be considered and modified. However, there are defects which are fundamentally more difficult. In the remainder of this section, we will discuss the **entity resolution problem (ER(P))**. In particular, we will look at the **author name disambiguation problem** which is a sub-instance of the entity resolution problem.

Entity resolution refers to the task of matching information to specific entities (for example, organizations, persons or countries). The author name disambiguation problem in digital libraries is the task to match an author name to a person entity. There are two fundamental problems:

- Multiple authors can have the same name. Such a name is called a **homonym**.
- A single author might use different names in different publications. The different names are called **synonyms**.

Figure 4.1 shows a combined example. The real world author entities *James Doe* and *James A. Doe* both publish as *Jimmy Doe* which makes this name a homonym. However, *James Doe* also publishes as *J. Doe* which makes *Jimmy Doe* and *J. Doe* synonyms. Users searching for *Jimmy Doe* will obtain publications from two different authors. On the other hand, some publications of *James Doe* are missing from this list. Apart from search-related issues, a wrong author entity can affect the author's reputation, especially in academics. Henzinger et al. [HSW10] note that disambiguation issues can negatively affect the omnipresent h-index [Hir05].

Schulz [Sch16] uses Monte Carlo simulations to study the influence of data quality on rankings of scientists. He finds that even slight deteriorations in the quality of person ↔ publication mapping can have a significant influence on the result. Wrong attribution of papers can translate to wrong assessment of institutions. Many countries base institution founding partially on publication performance [Hic12]. Laender et al. [LdLM+08] use data from digital libraries to assess the quality of Computer Science programs. These applications require correct mapping of persons to institutions but also correct mapping of publications to persons. Note that synonyms and homonyms are not limited to personal names but can be found for all named entities like place names and organization names. Crane and Jones [CJ06] present the particularly striking example of the name *Virginia Banks* that might be a town, a person, an organization or a geographical feature.

There are multiple reasons for homonyms and synonyms. Homonyms naturally occur if two authors have the same name. For example, in March 2017, DBLP differentiated between about 110 persons with the name *Wei Wang*[1]. The problem is intensified when incomplete names are provided, e.g., when the given name is abbreviated. In scientific publications, this practice is still common. The DBLP project obtains most publication data directly from the publishers. Of all author names provided between 2011 and 2015, 12.8% had the first name abbreviated. Abbreviated names also contribute to the synonym problem. An author might be referred to by the abbreviated name and the full name. Person names can change over time for example after a marriage or a divorce. Synonyms can also occur because the use of different transcription systems (e.g., Vladimir – Wladimir), inconsistent omissions of name parts, encoding issues (e.g., Mueller – Müller) [Reu07].

There are different strategies to deal with synonyms and homonyms.

- Use of unique identifiers.
- Use of authority files.
- Heuristics which determine the author.

A unique identifier would be provided in addition to the author name and uniquely identify the person entity. However, this approach will only work if all (or at least a significant portion) of the authors use them. Enserink [Ens09] points out that a single authority is needed to provide such identifiers. ORCID[2] might achieve this goal. At the moment of this writing, actual publisher support is limited but growing rapidly. Authority files provide identifiers within the scope of the library. For each person entity there is a record that provides distinguishable features, e.g., date of birth, profession, and a way to reference the person. They also provide a list of all known names which have been used by the author. Many libraries use an annotated name to refer to entities in the authority file, e.g., *John Doe, 1960-*. Table 4.1 shows

[1] `https://dblp.org/pers/hd/w/Wang:Wei` retrieved March 2017
[2] `http://orcid.org`

Table 4.1: Authority identifier used by different projects.

collection	dist. feature	reference example
DBLP	consecutive number	Wei Wang 0010 – Wei Wang 0011
IMDB	consecutive number	John Smith (I) – John Smith (II)
Wikipedia	bio. properties (profession, lifetime, nationality ...)	Anil K. Jain (computer scientist, born 1948) – Anil K. Jain (electrical engineer, born 1946)

how some of the collections we discussed in Section 2.4 handle these identifiers. Creating and maintaining authority files is time-consuming. The national libraries have created large authority files, e.g., the *Library of Congress Name Authority File* or the *Gemeinsame Normdatei* by the German National Library. However, these files are not complete and a library might still need to create their own authority records. It is also still necessary to map authors of new publications with the authority file.

As identifiers are not widely available and authority files are expensive, algorithmic disambiguation solutions are popular. A disambiguation algorithm uses properties of records and possibly additional information (e.g., results of web searches [TKL06]) to determine the actual author entities. In the example of Figure 4.1, an algorithm might differentiate the two person entities because they publish on different topics or in very different years. We will discuss some of these algorithms in more detail in Section 4.3.1.

4.2 Detecting Past Defects

Our goal is to extract a significant number of historical defects from real-life digital libraries. We will study the properties of the defects and create test collections for ER algorithms. Traditional approaches use massive manual work to find defective records. In this work, we harness the manual work which has already been invested in the collections as part of the normal curation process. This approach is an extension of the work of Reuther et al. [RW06][Reu07]. In their work, the authors compare the state of DBLP at two different times. In particular, they check if an author element of a record has changed. They then checked if the old author name was present at all in the new state of the collection. If not, they considered the modification to be the correction of a synonym defect. From these detected corrections they created a test collection which can be used to evaluate ER algorithms.

We will extend this approach in two ways: (1) we will consider modifications for all consecutive states instead of comparing two versions. This will result in more corrections. We will also obtain an order of corrections and find defects that needed multiple (partial) corrections before they were resolved. This information can be used to analyze error correction strategies of a project. (2) We also consider corrections of homonyms and a mixed form which we will call *distribution* defect. In the

following section, we will discuss how to automatically identify defect corrections in our framework. This approach is not without bias. In Section 4.2.2, we will discuss this in detail. Section 4.3 discusses how to create test collections from corrections.

4.2.1 Inferring and Classifying Corrections from Historical Metadata

Vivid digital library projects are constantly growing and changing. New records are added and others are completed with new data elements, which have become available. There is also a large number of legitimate reasons to modify a data element which are not related to defective data. Our goal is to automatically filter out these modifications so that we retain the corrections of defects. In this work, we consider two types of corrections: **value corrections** and **reference corrections**.

A value correction repairs an isolated data element by removing a defective value and optionally adding a non-defective one. Adding is optional as we must consider the case where defective data is deleted without replacement. There are a number of scenarios in which we can detect value corrections:

(a) A data element which is considered immutable is modified.

(b) The value of the data element is illegal.

(c) The value is inconsistent with values of other data elements.

(d) The value violated a uniqueness constraint.

Few metadata definitions define a strict immutability of data elements. However, in practice, some data elements of a collection are normally not subject to changes. Consider the following example based on a DBLP record:

Example 4.1:

```
year: 2013
ee: http://example.com/paper10
```

year holds the publication year of a record while *ee* stores a link to an external resource. If we analyze the DBLP collection, we see that *ee* elements are changed frequently. This is often the case when publishers change the structure of their website. On the other hand, *year* information are changed very rarely. As the publication year cannot change externally, we must assume that either the old or the new year is wrong (or possibly both). Knowing which values are immutable requires domain knowledge on the collection.

The same knowledge is required for cases (b) and (c). We must be aware of explicit (through metadata definitions) or implicit (through usage) constraints on the data. Assume a record stores the ISBN *0262527138*. We observe an edit which replaces this number with *978-0-262-52713-2*. In this case, an ISBN-10 was replaced with an equivalent ISBN-13. For most purposes this is not a correction, although there might be scenarios where we would consider an ISBN-10 outdated or inconsistent. To determine that this modification is not a correction, we need to know that the two strings are equivalent for most purposes. Note that consistency checking at the time of entering the data is not as simple as for classical database systems. While most metadata management systems provide some consistency checking [BBMU11], it is often necessary to allow illegal data. For example, a *year* element should hold a number, but it might also hold a value like *before 1980* or *ca. 1985*. Uniqueness constraints (case (d)) are simple to detect. However, we can assume that most digital libraries use systems which prevent such violations. E.g., a metadata management system would not allow that two records have the same primary identifier. Summing up, automatic detection of value corrections is limited to specific cases where sufficient domain knowledge is available.

Reference corrections are not isolated but can affect a number of records at once. In Section 2.1.2, we introduced a view on metadata based on a graph. Instead of values and data elements, this view considers entities and their relations. We can combine this view with the record-based view. Consider the following data elements in a metadata record:

Example 4.2: Entity references in metadata record

```
author: Bob
publisher: Springer
```

In the graph-based view, we can choose to model persons and organizations. In this case, *Bob* would refer to a person entity and *Springer* would refer to an organization entity. We can define which data elements refer to entities.

Definition 4.1 (Signature, Surface Form): Let d be a data element. We call d a **signature** if d references an entity.

If d has an atomic value v which is a name, we call v the **surface form** of the signature.

We do not define how the data element references the entity. For some collections, the surface forms are used directly as identifiers. I.e., all signatures with the same surface form reference the same entity. Other collections attach identifiers to d or use an authority file on the surface forms. We restrict surface forms to names as those names play an important part in name disambiguation. In Example 4.2,

the surface forms are *Bob* and *Springer*. An entity can be referenced by different signatures at the same time.

Definition 4.2 (Theoretical Entity, Theoretical Reference Set): Let e be a real-world entity. We call e **theoretical entity** if the collection could contain signatures which reference e.

At a time t let $RT_t(e)$ be the set of all signatures in the collection that should reference e. We call $RT_t(e)$ the **theoretical reference set** of e at time t.

The theoretical entities and reference sets represent the ground truth of the collection. For non-trivial collections, theoretical entities and their reference sets are unknown. However, most collections use entity representations of their data. For example, DBLP provides **author profiles**. A profile contains all records with a signature that references this author entity. As the theoretical entities are unknown, the author profiles are an interpretation of the signatures in the collection. This interpretation can be wrong. E.g., DBLP primarily uses the surface forms for person entity identification. If two theoretical person entities have the same surface form (i.e., they are homonyms), their records might end up in the same author profile. The interpretation of signatures by the project creates a set of referenced entities which in practice is often not identical with the set of theoretical entities.

Definition 4.3 (Empirical Entity, Empirical Reference Set): Let E be a set of entities and S be a set of signatures which reference these entities. Let T be the set of observation timepoints.

Let function $I : S \times T \mapsto E$ be an interpretation of the signatures.

For $e \in E$ and a time t we call $RE_t(e) := \{s \in S | I(s,t) = e\}$ the **empirical reference set** of e at time t. We call $e \in E$ an **empirical entity** of the collection if $\exists t \in T : RE_t(e) \neq \emptyset$.

Function I represents the interpretation of signatures for the collection. This interpretation can change over time. I.e., a collection might first use surface forms to define empirical entities and later switch to an identifier-based system. For this work, we assume that the interpretation procedure is stable over time. This is the case for all collection we will discuss later.

Reference sets (theoretical and empirical) can change naturally over time. For example, a new signature might be added to the collection (i.e., a new document from a specific author was added). However, we can also find corrections of defects here. Assume that a collection used surface forms to define empirical entities. We already saw in Section 4.1 that names are unreliable identifiers. Assume that there are two theoretical entities e_1 and e_2 with surface form s (a homonym). The collection might interpret the signatures with this surface form s as referencing a single empirical entity e. e represents both e_1 and e_2 which is usually undesired and considered

a defect. In this work, we attempt to locate corrections to the empirical reference sets related to these defects. To fix the defect, the interpretation of the signatures in the project must change. There are two mechanisms that can change the entity referenced by a signature.

1. Changing the surface form or an attached identifier that is used for entity resolution. See Table 4.1 for examples.
2. Changing the authority file in a way that the mapping surface form $\leftrightarrow$ entity is affected. The data element itself is unchanged.

In case (1), there is a change to the corresponding data elements. E.g., DBLP will add a numeric suffix to the homonymous surface form (such as *Wei Wang 0020*) to keep the reference sets apart. This can be tracked by our modification framework. Modifications of type (2) are less obvious. Assume that one record contains a surface form *J. Doe* and another record a surface form *John Doe*. Assume further that both surface forms are supposed to reference the same person entity. A typical authority file allows adding of synonyms to persons. So once the similarity of *J. Doe* and *John Doe* is determined, we might create an authority record which links both surface forms. In this case, the data elements are not modified, only the interpretation of the data changes. For example, a retrieval engine will return records for both surface forms if one is queried. Case (2) is problematic as we often have no access to the authority file. Even if the data is available, it must be interpreted and integrated into the temporal modification framework. We will see in Section 4.3 how to handle the authority records of DBLP. Creating and maintaining an authority file is expensive and we can expect the file to be small for most collections so the loss of data is limited. In the remainder of this Section, we will concentrate on case (1) reference changes.

We can consider evolving reference sets from a combinatorial point of view:

> ***Definition 4.4 (Reference Successor/Predecessor):*** Let $t_1 <_{\mathcal{T}} t_2$ be two time points and let e_1, e_2 be empirical entities.
>
> We call e_1 **reference predecessor** of e_2 if $\exists\ s \in RE_{t_1}(e_1) : s \in RE_{t_2}(e_2)$. We call e_2 **reference successor** of e_1.
>
> We call e_1 **consistent predecessor** of e_2 if $RE_{t_1}(e_1) \subseteq RE_{t_2}(e_2)$.

In most situations, e_1 and e_2 are the same entity, i,e., an entity can be its own reference predecessor or successor. For two points of time, we can identify two candidates for reference corrections.

(a) An entity e has two or more reference predecessors.

(b) An entity e has two or more reference successors.

In case (a), e was represented by multiple empirical entities before. If we assume that e is correct now, the successors where synonyms. In case (b), empirical entity e has been split into multiple entities. This is the correction of a homonym defect. Cases (a) and (b) can occur naturally without being linked to defects. Assume that the entities represent organizations which can merge and split naturally in the course of time. These real-world modifications might eventually manifest in a situation described in cases (a) or (b). However, many entity types – such as persons – do not merge or split naturally. In these cases, we can assume that (a) and (b) represent the correction of a reference defect. I.e., the empirical entities we obtain are closer to the theoretical entities. For the rest of this work, we will focus on reference corrections for person entities.

Detecting and classifying corrections to references requires knowledge in the temporal relation of modifications across records. I.e., reference predecessors and successors might depend on modifications to multiple records. Therefore, we base the detection and classification of corrections on changes instead of edits.

Precondition: For the detection and classification of reference corrections:

- Collection must allow extraction of changes.

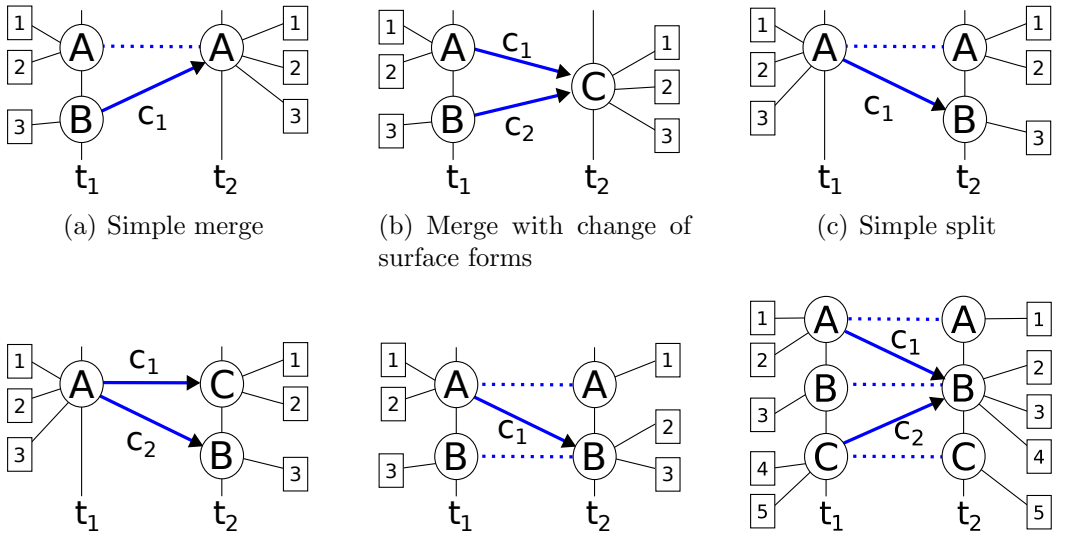
This excludes collections with local and logical clocks such as CiteSeer. We can categorize modifications to the reference set of persons as follows:

Definition 4.5 (Merge Group): Let e be an entity and $t_1 <_{\mathcal{T}} t_2$ two time points in the observation framework. Let $P := e_1, \ldots, e_k$ be the reference predecessors of e with respect to t_1 and t_2.

We call P a **merge group** if $k > 1$ and $\forall 1 \leq s \leq k : RE_{t_2}(e_s) = \emptyset \vee e_s = e$.

Between time t_1 and t_2, the signatures referencing $e_1, \ldots, e_k$ were modified so that they now reference e. These entities, except e itself, do not have any reference left. If the change was correct, the entities, that are now without reference, were pseudo entities that represented only a part of a person's work.

A merge group is represented by a set of changes. For a merge group $e_1, \ldots, e_k$ we can find a change that transformed the surface forms which reference $e_s : 1 \leq s \leq k$ into surface forms which reference e (except for $e_s = e$ of course). We call these changes a **change cluster**. We call the old values of the changes **source values** and the new values of the changes **target values**. For a change cluster that is linked to a merge group, we can have multiple source values but only a single target value. Figures 4.2(a) and 4.2(b) show typical examples of the change group of a merge cluster.



(a) Simple merge

(b) Merge with change of surface forms

(c) Simple split

(d) Split with change of surface forms

(e) Distribute with one change

(f) Distribute with multiple changes

Figure 4.2: Examples of corrections. Letters denote entities, numbers denote signatures. c_x denote changes.

Merges correct defects related to synonyms. We can define a similar correction for homonym related defects:

> ***Definition 4.6 (Split Group):*** With e, t_1 and t_2 as in Definition 4.5. Let $P := e_1, \ldots, e_k$ be the reference successors of e with respect to t_1 and t_2.
>
> We call P a **split group** if $k > 1$ and $\forall 1 \leq s \leq k : RE_{t_1}(e_s) = \emptyset \vee e_s = e$.

A split creates new empirical reference sets. The empirical entities references by these sets were fully submerged in pseudo entity e. Figures 4.2(c) and 4.2(d) show the base cases of a split.

For a merge, we demand that the merged entities are no longer referenced in the library. Similarly, we demand for a split that *new* entities have no signatures referencing them at t_1. In addition to that, we need to consider a combination of merge and split:

> ***Definition 4.7 (Distribute):*** Let c be a change which replaces references to e_1 with references to e_2. Let t_1 be the last time of the observation before c and t_2 the time of observation of the last edit of c.
>
> If $RE_{t_1}(e_1) \neq \emptyset$, $RE_{t_2}(e_1) \neq \emptyset$, $RE_{t_1}(e_2) \neq \emptyset$ and $RE_{t_2}(e_2) \neq \emptyset$, we call c a **distribute**

Table 4.2: Number of classified change clusters by project.

	merge	**spit**	**distribute**	**rename**	**total**
DBLP	110,697	11,241	42,052	52,747	216,737
IMDB	41,172	9,576	33,498	141,917	226,163

Figure 4.2(e) shows a distribute where change c_1 changes some of signatures which referenced A so that they now reference B while both A and B existed before and after the change. Distributes are different from merges and splits as both entities are represented before and after the operation. An algorithm which aims to correct this defect must determine which signatures are to be reassigned. If both reference sets exist before the correction the algorithm can use properties of A and B to determine which signatures to change. The change cluster of a distribute consists of a single change. However, multiple distributes might overlap as shown in Figure 4.2(f). For now, we consider these distributes as different corrections.

Finally, we need to consider consistent changes of signatures.

> ***Definition 4.8 (Rename):***
> If e_{old} is consistent predecessor of e_{new} and e_{old} is the only reference predecessor of e_{new}, we call $\{e_{old}, e_{new}\}$ a **rename**.

A rename changes all surface forms that reference an entity without changing the reference set itself. In a collection where entities are identified through their name, this might occur if the name is consistently changed. E.g., the surface form *A. Jones* is replaced with *Adam Jones* in the entire reference set. In the context of this work, we do not consider this to be a correction as the entity was correctly identified.

We applied extraction and categorization to DBLP and IMDB. Table 4.2 shows the number of classified changes per data set. In the following sections, we will primarily work with these information.

4.2.2 Threats to Validity

In the remainder of this chapter, we will use extracted reference corrections to study properties of defects and to create test collections. Both applications require that the detected defects are representative for the whole collection and not significantly biased. Our method for detecting defects has a number of intrinsic biases which cannot be fully mitigated. In this section, we will discuss the most relevant points. Each of these threats to validity must be considered before undertaking a study based on the historical defect corrections.

Assumption: Changes improve data quality. We assume that a correction replaces defective element values with correct ones. Obviously, there is no guarantee for that as the changes related to the correction can also introduce errors. The likelihood of introducing new errors depends on the data curation process of the individual projects. The collections we discussed in Section 2.4 have very different processes. DBLP and IMDB are similar to classical libraries where corrections are performed by a trained team. For such projects, we can assume that corrections are undertaken with some care and the chances of adding defects are relatively small. In particular, we can assume that corrections are performed with more care than the routine adding of new data. For both projects there is also an active community which reports defects, in fact, IMDB started with user-created content. In [RH11], we showed that for DBLP, users are able to contribute with acceptable quality. CiteSeer relies heavily on algorithms to find defects but also permits indirect user input. OpenLibrary and Wikipedia are crowd-based data sets. I.e., curation of the content is in the hands of the users. Ideally, the community would ensure that modifications are correct. However, different types of vandalism – unlike for closed projects – are a significant problem [SH10].

Assumption: Corrections completely remove defects. A correction might remove a defect only partially. Assume that an empirical reference set contains five signatures to real-world author A and another five references to author B. A correction (a split) might extract three signatures but leave two signatures behind. The original profile is still a homonym. If we build a test collection based on partial corrections, we must allow for a case where an algorithm finds the whole correction. We need to define specific evaluation metrics to handle this situation. We discuss this in Section 4.3.5. We will also present an analysis on the properties of corrected reference sets before and after the correction. During this analysis, we must account for a significant amount of partial corrections.

Assumption: The corrected defects are representative for the set of all defects. Our approach is biased by the way defects are detected in the underlying data set. Assume that a project applies a process which is good at finding defects with property $\mathcal{A}$ but can barely handle defects with property $\mathcal{B}$. In this case, defects with property $\mathcal{A}$ would be overrepresented and many defects with property $\mathcal{B}$ would be missing. We will later argue, that many algorithmic solutions might fail when only small amounts of data are available, e.g., for authors who have few publications. Projects which heavily rely on automatic detection might not have corrections for these authors. It is also possible that a project is aware of a defect but does not fix it because it has a low priority. Again, it is unclear how community contribution can mitigate this problem. For all studies, we must assume that error classes exist that are significantly underrepresented.

Assumption: The historical framework creates acceptable edits and changes. In Chapter 3, we discussed the principal limitations of the historical modification framework. For value and assignment correction, we consider the modification of values. Assume that a data element is modified in a way that the value

is changed significantly. Depending on the settings of the value comparator, we might consider this as a modification edit or as a set of an add and a remove edit. As corrections are computed based on changes which are based on edits, we might miss out the correction. This work concentrates on reference corrections related to person entities. We adjusted the heuristics which compute the edits so that they cover the most relevant changes related to name modifications. However, we will miss complete changes of names.

4.3 Test Collections Based on Extracted Corrections

Table 4.2 shows that we can extract a significant number of corrections from DBLP and IMDB. In this section, we discuss how test collections for ER algorithms can be created from these corrections. In particular, we will create two test collections based on corrections in DBLP.

A test collection contains a set of signatures. A second data set contains the *ground truth* or *gold standard*, i.e., the theoretical entities and reference sets for the signatures. The algorithm is judged on how good it can approximate the ground truth. There are different ways to create the ground truth of a test collection:

- **Manual cleansing**: a data set or parts of it are thoroughly analyzed. This usually includes manual confirmation of data, e.g., by contacting authors of publications. Data that cannot be confirmed is removed from the test collection.
- **Pooling**: The results of different existing algorithms are combined. For example, the majority result is considered the gold standard.
- **Artificial collections**: defects are introduced by the creator of the test collection. The goal of algorithms is not so much reconstructing a gold standard but finding those defects.

All approaches have advantages and disadvantages. Manual cleansing is time consuming and therefore often only yields small test collections. While manual cleansing will probably produce the most reliable gold standard, errors will occur. Pooling can create large test collections for many different projects. However, there is no guarantee that the algorithms will create a reliable ground truth. Even if one algorithm is able to handle a certain defect, it might be outvoted by the other algorithms. Using artificial defects allows the creator to add defects with certain properties. However, the problem is to ensure that these properties are realistic, i.e., the evaluation results can be reproduced on realistic collections.

In this section, we discuss how information on corrected defects can be leveraged to create large and realistic test collections. We present two different test collections.

The case-based collection (see Section 4.3.3) consists of small test cases which are directly derived from the detected defect corrections. The embedded collection (see Section 4.3.4) provides a single test case which combines corrected defects over a longer period of time. Harnessing previous corrections for the test collection creation has two advantages.

- It combines the curation effort of a project with minimal extra costs. Thereby, it emulates the work done in a manual cleansing approach.
- It can be applied to any collection that allows tracing of historical metadata and the extraction of changes. I.e., domain-specific test collections can be created.

We apply our approach to the DBLP collection. We choose DBLP because it is the collection for which we have the best understanding of content and data handling. DBLP is also published under ODC-BY 1.0 license[3] which makes distribution of derived test collections simple. In Section 4.3.1, we will discuss state-of-the-art ER algorithms. This is necessary to understand which data is needed from a collection and what scenarios need to be evaluated. In Section 4.3.2, we discuss test collections that are currently available and how they are used. We will see that the two collections we develop are different from the existing ones in a way that they cannot be used in typical evaluation scenarios. In Section 4.3.5 we propose evaluation scenarios for the collections. Besides a possible use as a test collection, the extracted defect corrections can also be used to understand properties of defects. We will use the case-based test collection to study surface form-based properties (Section 4.4) and network-based properties (Section 4.5). Both collections are published under the open ODC-BY licence[Rei18].

4.3.1 Overview on ER-Algorithms

In this Section, we describe current approaches to deal with the entity resolution problem (ER). We focus on two aspects which are relevant for our work: application scenario and data use. There are two fundamental scenarios in which these algorithms are used:

- Batch resolution
- Defect detection and iterative adding (sometimes called online disambiguation task)

Batch resolution algorithms process the signatures of the entire collection. Let $S := s_1, \ldots, s_k$ be all signatures in a collection. Batch resolution algorithms compute a partition $P := p_1, \ldots p_s$ of S. Ideally, each p_i is identical with a theoretical reference set. The goal is to minimize homonyms and synonyms in P.

[3] `http://opendatacommons.org/licenses/by`

Some approaches define pairwise similarities between signatures and use clustering algorithms to group them (e.g. [HZG05], [LIDK+14]). Other groups consider a classification problem where a classifier decides to which reference set a signature belongs (e.g. [WBH+12], [HGZ+04]). The literature contains some variations of batch resolution. E.g., D'Angelo et al. [DGA11] try to map the signatures in a digital library to a list of researchers which is provided by a government organization. They ignore all references to authors who are not in that list. Batch resolution can ignore any empirical entities that already exist at the time of computation. This ensures that previous errors are not propagated. However, this can cause signatures to oscillate between two reference sets at every run. Also, may approaches do not allow to preserve reference sets which are known to be correct, e.g., because they were manually checked. Batch resolution can be algorithmically expensive as the complete collection must be processed. Very few approaches are evaluated against collections with the size of DBLP. Nevertheless, the majority of algorithms we present in this section use batch computing.

In the *iterative adding* scenario, a single signature is added to the collection (usually, as part of a new record). Given a collection with reference sets $P := p_1, \ldots p_s$, the algorithm has (1) to determine the set to which the new signature belongs or (2) to add it to a new reference set. Unlike a batch resolver, an iterative algorithm cannot ignore existing reference sets. Therefore, many iterative algorithms are able to handle reference errors in the collection. E.g., Santana et al. [SGLF15] check for reference sets which need to be merged after the new signature has been added. Some iterative approaches can also be used for batch resolution. E.g., this is explicitly described by Santana et al. [SGLF15]. Qian et al. [QZS+15] propose a system that performs a single large batch entity resolution followed by sequence of partial resolutions. These partial resolutions can be quite small and effectively make the approach iterative.

Regardless of the application scenario, ER algorithms have to deal with the small amount of information which is available in bibliographic collections. We can differentiate between three types of information.

- The local record information.
- The relational data between entities in the collection.
- External data, obtained from outside of the test collection.

Table 4.3 lists the data used by 40 ER algorithms. In the table, we only note data which is explicitly mentioned in the publications. Most approaches can be extended to use other information as well.

Local record information refers to all information which is directly available from a bibliographic record. The nature of this information depends on the collection.

Some information are record specific and identical for all of its signatures. For bibliographic data, these are usually the publication title, venue[4] and date of publication. Data like the person's name (the surface form), affiliation, email addresses and so on are specific for a signature, e.g., two authors in the same record can have different affiliations.

Local record data are often used for **blocking**. All ER algorithms need to consider a similarity between signatures or reference sets. This is often done for all pairs of signature/reference sets. These similarity functions and the clustering/classification built on top of them are often too expensive to be applied to the whole collection. To overcome this problem, the data is divided into blocks. The goal is to create blocks which are small but contain all relevant information. E.g., a block should contain all signatures related to a theoretical reference set. The more expensive functions are then applied for each block individually, reducing the number of pairwise comparisons. Blocking functions need to be fast. Therefore most approaches are based on the local record information which is readily available. Yan et al. [YLKG07] present an approach that uses adaptive windows to create blocks of different sizes. For their experiments they use the first characters of a person's last name or a combination of the first name initial and the last name. On et al. [OLKM05] also experiment with blocking based on the last name. Bilenko et al. [BKM06] present an adaptive blocking which uses most of the local attributes. Evangelista et al. [ECdSJ10] use genetic programming based on most local attributes. Louppe et al. [LASM16] use a similar approach but also consider phonetic variations of the names as well as differences in name representation. They also include cultural aspects of the name for blocking.

The local record data is also used to compute the **similarity** of signatures or reference sets. Most early approaches mainly use author name similarity as evidence. Assume that an entity is referenced through two different and unmatched surface forms. While it is possible that the two surface forms are completely different, they are more likely to show some kind of similarity. In a collection with surface forms *J. A. Doe*, *John A. Doe* and *Frank Miller*, we might consider *J. A. Doe – John A. Doe* to be synonyms but not *J. Doe – Frank Miller*. Many different similarity functions for personal names have been proposed. Cohen et al. [CRF03] and Elmagarmid et al. [EIV07] compare different approaches for the entity resolution task. Bilenko et al. [BMC+03] discuss string similarities which adapt to a specific collection. Gong et al. [GWO09] use a different approach to find synonyms by transforming names according to specific rules. Similarity is measured by the number of transformations that are necessary. Treeratpituk and Giles [TG12] propose a system that determines the cultural origin of an author's name. This information is used to apply culture-specific similarity functions. Louppe et al. [LASM16] use a similar approach.

[4]We use the term *venue* to refer to any outlet such as journals, conferences or other forms of publication series.

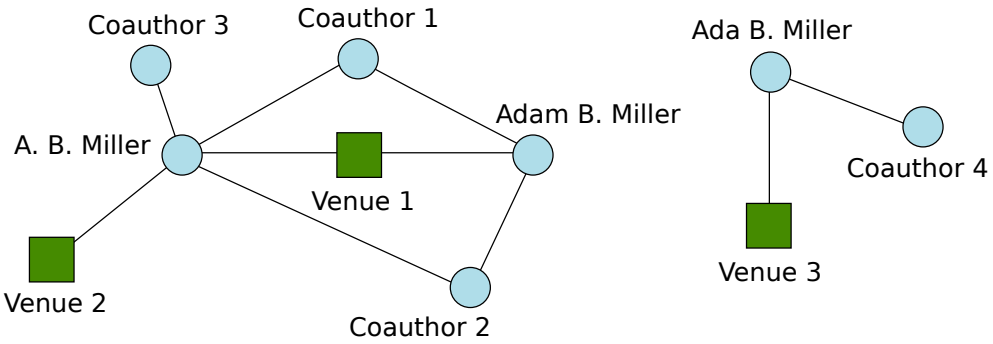


Figure 4.3: A network of entities and relations typically found in bibliographic data. The network structure suggests similarity of *A. B. Miller* and *Adam B. Miller*.

Most modern approaches consider multiple types of local record data. Figure 4.3 shows the basic idea on a synonym detection example from scientific publishing. The surface forms *A. B. Miller* and *Adam B. Miller* are similar for some string similarity function but assuming that both surface forms reference to the same person entity is dangerous. I.e., for reasonable large collections, it is possible that *A. B. Miller* is unrelated to *Adam B. Miller* (perhaps referencing an *Ada B. Miller*). However, *A. B. Miller* and *Adam B. Miller* not only have similar names, they both collaborated with *coauthor 1* and *coauthor 2*. Also, they published papers on the same venue. *Ada B. Miller* is in no such relation to the other two surface forms. An algorithmic synonym detector might use *coauthor 1*, *coauthor 2* and *venue 1* as evidence for the identity of *A. B. Miller* and *Adam B. Miller*. Note that Figure 4.3 uses a graph-based presentation. However, most approaches consider the properties as strings that are used in bags of words or feature vectors. For example, Santana et al.[SGLF15] consider the coauthors of a signature and the venues of publications as a set of strings. These sets are used to compute a similarity score of reference sets. Most approaches can be extended to incorporate additional data. E.g., Culotta et al. [CKH+07] use words extracted from publication titles as indicators if the publications are from the same author. Their approach could easily be extended to words extracted from publication abstracts. However, most test collections do not provide abstracts so this feature is not explicitly explored.

Signature-specific local data, such as affiliation and email address, can greatly support the author disambiguation. Zhu et al. [ZYX+14] make use of affiliation and grant number to compare signatures in PubMed. They note that these information is available for 53.2% (affiliation) and 8.5% (grant) of all signatures. Wu et al. [WD13] use affiliations and coauthor names to track scientists with changing institutions. However, relatively few approaches use those data. As before, the likely reason is that most established test collections do not provide these data and testing is not possible.

Table 4.3: Data used in 40 ER algorithms. For publication that do not specifically state the local data they use we assume name, coauthors, title, venue and date.

symbol	description
	Record
A_n	name surface form or parts of it: [BG07], [CFN+10], [CKH+07], [DGA11], [FVGL10], [FVGL14], [KMP07], [HEG06], [LKBJ12], [LIDK+14], [LASM16], [QHC+11], [QZS+15], [SGLF15], [SKCK14], [SLM09], [SHC+07], [TFWZ12], [TS09], [TG09], [VFG+12], [WBH+12], [ZDH16]
A_{email}	author email: [CKH+07], [LKBJ12], [QHC+11], [TS09]
A_{affil}	author affiliation (institute name, address ...): [CKH+07], [DGA11], [HEG06], [LIDK+14], [QHC+11], [TS09], [TG09], [WD13], [WLPH14]
R_{co}	direct coauthor names: [BG06], [CFN+10], [CKH+07], [FVGL10], [FVGL14], [HGZ+04], [HXZG05], [HZG05], [KMP07], [LKBJ12], [LASM16], [QHC+11], [QZS+15], [SGLF15], [SLK16], [TG09], [VFG+12], [WD13], [ZDH16], [ZYX+14]
R_{ti}	record title: [CFN+10], [CKH+07], [FVGL10], [FVGL14], [HGZ+04], [HXZG05], [HZG05], [HEG06], [LIDK+14], [LASM16], [QHC+11], [QZS+15], [SGLF15], [SHC+07], [TFWZ12], [TG09], [VFG+12], [WBH+12], [WLPH14], [ZDH16]
R_{ve}	record venue name: [CFN+10], [CKH+07], [FVGL10], [FVGL14], [HGZ+04], [HXZG05], [HZG05], [HEG06], [LKBJ12], [LIDK+14], [LASM16], [QHC+11], [QZS+15] [SGLF15], [TFWZ12], [VFG+12], [WLPH14], [YPJ+08], [ZDH16]
R_{date}	publication date, usually the publication year: [DGA11], [LIDK+14], [LASM16], [QHC+11], [QZS+15], [SGLF15], [TFWZ12], [TG09]
R_{lang}	record language: [LKBJ12], [TG09]
R_{key}	controlled vocabulary such as keywords: [DGA11], [LKBJ12], [LIDK+14], [QZS+15], [SHC+07], [TG09], [WBH+12]
R_{abst}	abstract from the publication: [LKBJ12], [LIDK+14], [LASM16], [QHC+11], [QZS+15], [SHC+07], [TFWZ12], [WBH+12], [WLPH14]
R_{cit}	list of works cited by this record (from the bibliography section): [LKBJ12], [QHC+11], [TFWZ12], [WBH+12], [WLPH14]
	Network
G_{co}	full coauthor graph: [BG07], [FWP+11], [HYM+16], [LH10], [OEL+06], [SKCK14], [SSK+16], [TFWZ12]
G_{cit}	citation graph: [TFWZ12]
	External
E_{text}	document full text: [SHC+07]
E_{bio}	author biography extracted from documents or web pages, this includes publication lists: [HYM+16], [WLPH14]

E_h	human worker, either domain experts or persons participating in crowd source system: [QHC+11], [DDC13]
E_{query}	results of a web query based on local record data: [KMP07], [PRZ+09], [TKL06], [YPJ+08], [ZYX+14]
Others	
[LIDK+14] (grant, substance), [QHC+11] (author homepage URL), [HZ09] (Wikipedia)	

Many local data types require preprocessing before they are used. E.g., Zhang et al. [ZDH16] describe an approach where records are added iteratively. A record is transformed into a vector that contains authors and terms extracted from title and venue name. A Bayesian classifier is used to assign the publication to an existing reference set or to create a new one. Culotta et al. [CKH+07] (among other data) use words extracted from publication titles as indicators if the publications are from the same author. Qian et al. [QZS+15] propose a combination of batch and iterative ER algorithms. For titles and abstracts, they remove stopwords and extract bigrams which are used in feature vectors. Many approaches use similarity measures which require data from all records of the collection. Qian et al. [QZS+15] normalize their feature vectors with an *inverse paper frequency* which is similar to classical IDF (inverse document frequency). Qian et al. [QHC+11] introduce a similar measure called *Shared item frequency*. Loupe at al. [LASM16] use classic TF-IDF on most local record data (such as author name, title and abstract). The goal of these normalizations is to reduce the weight of common words. While inverse document frequency can be computed with local data, it still requires knowledge of all (or many) records of the collection. Small test collections are problematic for these measures as document frequency computed on a small sample can be meaningless. Some approaches require the computation of a model. E.g., Shu et al. [SLM09] use local record data to create a topic model. The model is later used to describe and compare the research area of publications in reference sets. Models might not be able to use the data from the whole collection. Shu et al. note that DBLP is too large for their approach (with computers available in 2009). However, a significant number of records is needed and although models can be reused, it is not clear if they can be transferred from one collection to another.

Local record data can be scarce. I.e., a single record provides only very limited information. One strategy to increase the amount of available information is to consider local data from related articles. Levin et al. [LKBJ12] make use of all available record properties. In addition, they consider the local properties of related articles. E.g., they consider titles and abstracts of cited articles. Another approach to obtain more data is to harness the **relational data** which can be found in the collection. We saw in Section 2.1.2 that metadata in collections can be considered as a graph. Structures found in this graph can be used to draw conclusions on the grouping of references. E.g., Fan et al. [FWP+11] use a graph where each node represents a reference set. Two nodes are connected if the entities referenced by the sets coauthored a paper. Fan et al. use paths in this coauthor graph to compute

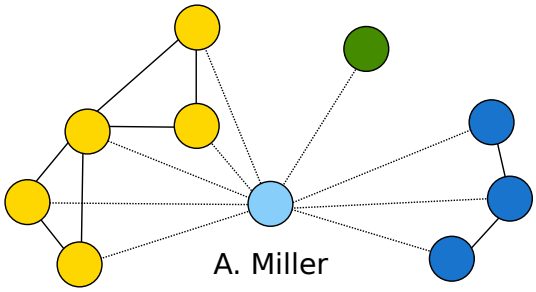


Figure 4.4: A local coauthor graph surrounding *A. Miller*. Edges denote coauthor relations. Colors denote connected components of this graph if the central node is removed.

the similarity between reference sets. They consider the shortest path between two nodes as well as the total number of distinct connections. This approach requires the full coauthor graph, instead of comparing lists of local coauthor names. Approaches that want to use graph data must extract these information and store them in a way that they can be queried in short time. Fan et al. notice that most of their paths are short, but in theory they can encompass the whole graph. Limiting the graph is difficult. E.g., the coauthor graph of DBLP is a small world network which means that the average distance between pairs of connected nodes is small. Sun et al. [SSK+16] use a similar graph. They use random walks to compute the similarity between the nodes. Starting at node s, the walker traverses the graph. The likelihood of ending the walk at node t defines the similarity between s and t. Sun et al. then use graph clustering algorithms based on the similarity defined by the walker. The walkers can be configured with a parameter c which is the probability of returning to the starting node. c controls the distance a walker can travel from s. It can be limited to the immediate coauthors or to include the whole reachable graph. Sun et al. use a small value for c which permits travel in large neighborhoods. Graph structures can be used to detect homonyms. Consider the following simple algorithm used by DBLP. The general idea is that entities are embedded in their neighborhood but that the neighborhood is also connected in several other ways. Consider Figure 4.4, which shows the coauthor neighborhood of the node represents the empirical entity *A. Miller*. By construction, this neighborhood is connected through *A. Miller*, but there are other relations as well. If we remove *A. Miller*, we obtain three connected sets of nodes (denoted by color). I.e., *A. Miller* published with three unconnected groups of persons. Working with more than one community of coauthors is not uncommon. However, a coauthor graph as shown in the example might be an indicator that *A. Miller* references more than one real-world entity.

Besides coauthors, it is also possible to consider the citation graph. The citation graph contains an edge between two publications a and b if a formally cites b. Citation analysis is an important part of scientometrics and has also been explored for entity disambiguation. Qian et al. [QHC+11] use citation as part of a similarity measure between publications. Tang et al. [TFWZ12] use a citation graph that

contains the direct citations, but also edges between papers which are cited by the same paper. Note that tracking citations is difficult and error-prone and many collections do not provide them.

Another strategy to handle the scarce data problem is to include external sources of information. External data can support name disambiguation, but it also requires a reliable mapping between the collection and the external source. A common approach is to use the result of search engines. Local record properties such as author names and title are used to compose queries. Tan et al. [TKL06] generate one query for each publication. The likeliness that two papers are written by the same person is based on the number of URLs which is returned by both queries. Pereira et al.[PRZ+09] also use web queries but apply a heuristic which identifies single author publication lists. Such lists can be found in CVs and provide reliable evidence of the publications belonging to a person. Zhu et al. [ZYX+14] use learning-based approaches to determine if a web page is a personal page where the author provides a publication list. Kang et al. [KNL+09] present a similar approach where they combine names of two authors in a query for web pages. They search for co-occurrences of the names which they consider as implicit relations. Han et al. [HZ09] use network data obtained by linking the author names with surface forms in Wikipedia. Of course, this requires a significant coverage of authors in Wikipedia which is not the case for most digital libraries. Another approach tries to enrich records with more data. Shin et al. [SKJC10] mine publisher web pages for abstracts of scientific publications from which they extract the thematic interest of the authors. Similar approaches can be used to obtain affiliation data or emails. Hedeler et al. [HPM14] discuss using Linked Open Data (LOD) to enrich DBLP with data from GND and DBPedia. In this work, we consider the documents themselves as external data because (1) they require extensive preprocessing and (2) they are often not available for legal or logistical reasons. Some of the approaches described above use full text extracted from documents. Hazimeh et al. [HYM+16] propose to extract short biographies that can be found in some papers. These biographies can be used to track the historical affiliations of an author and thereby support comparison. They also briefly discuss using image mapping to the pictures provided with these biographies. Another aspect of external data is using human understanding to support algorithms. Qian et al. [QHC+11] present an algorithm that uses a combination of high precision and high recall clustering to create reference sets. Some of these profiles are then checked by human workers. The authors report that this hybrid approach increases the quality of the result significantly. Dimartini et al. [DDC13] propose a data integration tool which automatically generates crowdsourcing tasks (e.g. for Amazon Mechanical Turk) to match difficult cases.

4.3.2 Overview on Test Collections

ER algorithms are evaluated with test collections. As described above, a test collection consists of a challenge (a set of signatures) and a solution (the theoretical reference sets). An ER algorithm is considered good if the empirical reference set it

Table 4.4: Comparison of test collections. Properties: see Table 4.3. AM: Arnetminer: `https://aminer.org`, BDBComp: Biblioteca Digital Brasileira de Computacao `http://www.lbd.dcc.ufmg.br/bdbcomp`.

Cite	collection	properties	sign.
Han et al. [HXZG05]	DBLP	A_n (abbrev.), R_{co}, R_{ti}, R_{ve}	8,453
Culotta et al. [CKH+07]	DBLP	A_n, R_{co}, R_{ti}, A_{email}, A_{affil}, A_{abst}	3,007
Wang et al. [WTCY11]	AM	A_n, R_{co}, R_{ti}, R_{ve}, R_{date}	6,656
Kang et al. [KKL+11]	DBLP	A_n, R_{co}, R_{ti}, R_{date}	37,613
Cota et al. [CFN+10]	BDBComp	A_n, R_{co}, R_{ti}, R_{ve}	361
Quian et al. [QZS+15]	DBLP+AM	A_n, R_{co}, R_{ti}, R_{ve}, R_{date}	6,716
Reuther [Reu07]	DBLP	A_n, R_{co}, R_{ti}, R_{ve}, R_{date}	18,872
Momeni et al. [MM16]	DBLP	A_n, R_{co}	32,273
Müller et al. [MRR17]	ZbMath	A_n, R_{co}, R_{ti}, R_{ve}, R_{date}	33,810

Source Müller et al. [MRR17] and own work.

produces are similar to the theoretical reference sets. Test collections are also used to train algorithms. Algorithms that use supervised training require examples of correct data. As correct data is difficult to create, many approaches use parts of test collection solutions. In this section we describe test collections which are used for the evaluation of ER algorithms in the context of bibliographic metadata. We ignore test collections built of unstructured text such as the ER challenge collection [CCG+14]. The following section is a summary of the findings described in the survey of Müller et al. [MRR17]. The survey focuses on DBLP-related test collections but also presents a new test collection based on the zbMath[5] collection. Table 4.4 gives an overview on the test collections we consider here.

For most collections, the ground truth covers only a single signature per record. Coauthor surface forms are provided but are not part of the challenge or the solution. As coauthors are not fully modeled, there is no full network information. In particular, G_{co} cannot be reconstructed. This is a significant disadvantage for algorithms which use this property. Other local record data are provided as well. Most of the collections described by Müller et al. [MRR17] are based on DBLP. Therefore, the amount of available information is limited by the data provided by DBLP. However, Culotta et al. [CKH+07] enriched some of the DBLP records with author emails and affiliations as well as publication abstracts. Despite the small amount of available information from a DBLP record (see also Table C.1 for a list of the most common data elements in DBLP), the amount of provided information varies. E.g., Han et al. [HXZG05] provide only abbreviated names for the surface forms which are to be disambiguated. The test collection by Momeni and Mayr [MM16] contains only author surface forms. However, it provides the DBLP public-

[5] `https://www.zbmath.org`

ation key so missing data can be retrieved if needed. Table 4.4 lists the data which is directly available from the test collections. Test collections vary in size. Table 4.4 lists the number of signatures in the ground truth which ranges from 361 to 37,613. Even the larger collections are significantly smaller than DBLP or IMDB. The small size causes three problems. (1) It is unclear whether all relevant problems are sufficiently represented. (2) It is difficult to estimate the computational performance of an algorithm with such small collections. (3) Algorithms that need to train a model must make due with the small amount of available information.

Most of the collections in Table 4.4 are created by manual cleansing. Han et al. [HXZG05] was introduced in 2005. From DBLP, the authors created sets of names with identical first name initial and last name, e.g., *C. Chen*. They then determined the actual authors behind the signatures, discarding unclear data. Most of the other collections are created in a similar way, though as Müller et al. note, it is often unclear how the data cleaning step was performed. The collection created by Qian et al. [QZS+15] differs from the previous collections as it was created by pooling other test collections. However, the authors performed further cleaning on the pooling result using manual labelers. Kang et al. [KKL+11] describe a semi automatic procedure where an algorithm determines web sites of authors. The data from the web pages are then used for manual disambiguation. Manual cleaning can produce good results. However, it accounts for the small size of most test collections. There is also no guarantee that the collection is free of errors. In 2014, Shin et al. [SKCK14] concluded that 3.37% of the signature mappings in Han et al. [HZG05] were wrong and 22.85% could not be verified.

An alternative approach to create large test collections is to harness the disambiguation work which has already been invested in a collection. Reuther [Reu07] introduced a test collection that is created by comparing two states of DBLP. He extracts modifications to reference sets in DBLP and identifies synonyms. Momeni and Mayr [MM16] also make use of disambiguation work in DBLP. DBLP uses a numeric suffix to distinguish persons with the same name. For example, *Wei Wang 0001*, *Wei Wang 0002* and so on. Monemi and Mayr present a test collection which contains the signatures of these authors. They assume that disambiguations are done with care and that the reference sets of the disambiguated authors are correct. Müller et al. [MRR17] also present their own test collection which makes use of the disambiguation work of the zbMath collection and DBLP. Both collections overlap. Müller et al. compared the disambiguation work of both projects for this overlap. Based on their findings, they concluded that manually disambiguated parts of zbMath are reliable and sampled a test collection from this data.

Besides creating a test collection by cleaning existing data, it is also possible to artificially generate a collection. An artificial test collection provides a controlled environment for testing but modeling the properties of real collections is crucial. A simple approach is used by On et al. [OLKM05]. The authors select the 100 most prolific author profiles in DBLP with respect to the number of publications. Each author is split in half, i.e., two reference sets are created each with half of the signatures. For one of the sets a new surface form is created by abbreviating

the original name, adding typos and other strategies. On et al. use these pairs to test a synonym detector. As the authors note, this is not a realistic scenario and we will see in Section 4.5 that such large synonyms are very rare. It is also unclear how representative the types of name variations – the authors test two different distributions – are for DBLP. Ferreira et al. [FGA+12] present a tool (SyGAR) which harnesses information on property distribution from a real collection to create an artificial one. The tool simulates the likelihood of an author publishing with a new coauthor, at a new venue or on a new topic. One way to use such collections is to create a number of reference sets and group profiles with similar names. This is similar to the approach of Han but with artificial and controlled data. An algorithm then has to cluster the publications in such a group so that the correct reference sets are reconstructed. SyGAR can be parameterized to randomly add defects to metadata records. An example of a parameter is the probability that letters in a name are swapped or the probability that the first name is abbreviated. While many other parameters can be learned from the training collection, the defect parameters must be set by the user of SyGAR.

4.3.3 A Case-Based Test Collection

We will now discuss the creation of a test collection based on corrections to author entity references in DBLP. This test collection will contain an individual test case for each correction we observed. In Section 4.3.4, we discuss another collection which combines multiple corrections with the global state of DBLP at a certain time. The primary purpose of the case-based test collection is to study properties of defects in DBLP. In Section 4.4, we analyze surface form similarities of synonyms. In Section 4.5, we consider the network information which is available for each test case.

Let A be an empirical reference set which contains references to two different theoretical entities. Assume that we observe a split correction which creates a new reference set B from some of the signatures in A. We consider this a test case. For each test case we provide two files. The challenge file contains the unresolved defect directly before the correction. The solution file contains the state after the correction. I.e., we use the state after the correction as ground truth of the collection. Of course, this type of test is more limited than usual test collections. In particular:

- The correction might be incomplete. E.g., some of the signatures which should be in B remain in A.
- Further corrections could be necessary. I.e., A contains signatures of another person C which are not split away yet.

In addition, all the limitations we discussed in Section 4.2.2 apply here as well. However, the test collection which we generate contains a large number of test cases. Each test case is the result of a manual correction of the underlying collection. We can assume that the collection covers a wide range of cases which are usually ignored

in state-of-the-art approaches. For example, DBLP received multiple emails per day from researchers who request a correction to their publication list [RH11] (See Section 5.2 for more details). Often, the information provided by these mails cannot be found on the Internet or in the papers themselves. When building a classic test collection, these data would not be available.

Our goal is to provide as much data as possible with each test case. At the same time, we need to keep the size of the test cases manageable. We decided to provide challenge and solution as graphs instead of lists of records. This has two advantages. (1) The relational data are encoded directly into the collection. There is no need to reconstruct them from local data. (2) Graphs can provide context information which is difficult to integrate into a record-based representation. We identified three types of entities:

- **Document** represents the publications themselves.
- **Person** represents the authors of the publications.
- **Venue** represents the journal or conference where the paper was published.

DBLP provides unique identifiers for publications and venues[6]. The situation for persons is more complicated. As discussed before, authors in DBLP are identified through their name. However, there is a mechanism which maps surface forms to identifiers via specific authority records that are also subject to change. We can track the temporal development of these authority records. We created a temporal authority which can resolve a personal name to an identifier for a specific date. E.g., if surface forms *John A. Jones* and *J. A. Jones* are mapped via an authority record at the time of correction, we will represent them as a single node in the graph. When resolving historical names, we found that the publication records and the authority records are sometimes out of sync. This is because the authority records are created after the publication records have been added. I.e., a surface form might exist without an authority record to provide an identifier. We also found cases in which the historical DBLP file apparently missed revisions. This is the result of technical problems during the creation of the historical data set on which we base our observations. If a revision is missing, we might observe a surface form which has already been removed from the collection (e.g., by changing the value of an author element). In this case, a person identifier might not be available for the surface form as the authority record is already deleted. By comparing the record revisions with the authority records, we were able to identify most of the records with missing revisions. For the test collection, we removed all test cases in which a reliable person identification of the surface form is not possible. We removed 10% of all split, 2.2% of all merge and 3.3% of all distribute cases. The high number of rejected splits is caused by delayed creation of authority records for the entity which has not been

[6]The venue identifiers are defined on journals and conference level. This means that workshops and similar venues are often not fully modeled. See Section 5.1.2 for examples.

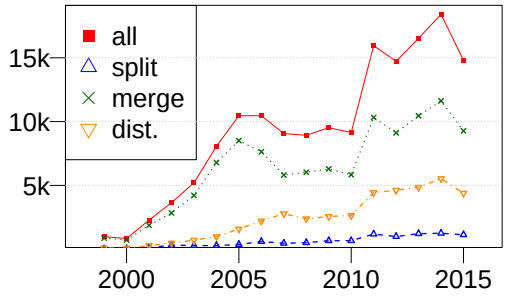


Figure 4.5: Number of retained test cases in the DBLP test collection by year and type.

referenced before. We obtain a collection with a total size of 158,992 test cases. Of these test cases, 108,229 are merges, 40,645 are distributes and 10,118 are splits. Figure 4.5 shows the distribution of corrections over time. The increase from 2010 to 2011 is caused by an increase in the team size of DBLP. For 2015, we obtained corrections between January and October. The actual number of corrections for that year is higher.

For a document node, we provide the most common data elements from DBLP. An overview of the properties can be found in Table C.1 in the Appendix. Surface forms of authors are not directly attached to document nodes but provided through relations (see below). We decided to use the most recent local values for documents instead of the values that could be found at the date of the correction. The primary reason is to provide the most recent *ee* links for records. The *ee* element stores a URL that points to an external reference site for the publications. This site is usually provided by the publisher. *ee* links can provide valuable external information such as author affiliation or document abstract. The links are subject to fast changes. In particular, many URLs that contain proper DOIs[7] were often added later and might not have been available at the time of the correction. Providing the most recent data increases the chances that the link still works. If a property appears multiple times in a record, the property stores a comma-separated list.

We model six different relations between the nodes. $\leftrightarrow$ denotes undirected relations, $\rightarrow$ denotes directed relations.

- **Created** (Person $\rightarrow$ Document): The person is author of that document. Unweighted.
- **Contributed** (Person $\rightarrow$ Document): The person has edited the document. Unweighted.
- **Co-Created** (Person $\leftrightarrow$ Person): The persons are author of at least one common paper. Weighted by number of common papers.

[7] `http://www.doi.org`

- **Co-Contributed** (Person ↔ Person): The persons have edited at least one common paper. Weighted by number of common papers.
- **Created-At** (Person → Venue): The person is author of a paper that appeared at the venue. Weighted by number of papers.
- **Contributed-At** (Person → Venue): The person edited work of venue. Weighted by number of papers.

Weights are computed for the last date before the correction was observed. I.e., the weights represent the data which would have been available to an algorithm at that time. We decided to model editorship and authorship separately as they might have different implications for an algorithm. Coauthorship usually implies cooperation while being coeditors (e.g., of a proceedings) can simply mean that the authors are active in the same field. The created and contributed edges show which surface form is stored in DBLP for this specific publication and the position in the author/editor list. Example 4.3 shows a *created* edge between the author with the ID *homepages/17/5330* and publication with ID *conf/wsc/RandellHB99*.

Example 4.3:

```
<edge source="homepages/17/5330" target="conf/wsc/RandellHB99"
      label="CREATED" directed="true">
   <property key="name" value="Lars G. Randell"/>
   <property key="position" value="1"/>
</edge>
```

As stated before, several surface forms can be attached to the same person identifier in DBLP. The property *name* is used to encode the surface form used for publications *conf/wsc/RandellHB99*. The combination edge and property can preserve the original value and express the relation of person and publication.

As stated before, the individual test cases need to be small enough to handle them. Obviously, we cannot provide the full metadata graphs (e.g., coauthor graph) for each test case. We decided to provide the local record information, as well as a limited context. For each graph we provide:

- The nodes which represent the empirical entities that were affected by the change (the primary nodes).
- All nodes directly adjacent to the primary nodes (regardless of the direction of the connecting edge).
- All edges between those nodes.

Providing edges between the nodes ensures that a minimal context is available. Assume that the author represented by primary node n has two coauthors c_1 and c_2. These coauthors might be related through documents which are not part of this test case. However, this relation might be useful for an ER algorithm. The graph-based view allows us to encode this information directly into the data.

The graphs of the test collection are provided as XML files. We call the collection which contains all test cases as described above **case-based-all**. We will later see that many test cases are small. E.g., a merge between reference sets with a single publication each. This is not surprising as it is not likely that large synonyms will live long without correction. In the same way, we consider splits or distributes small if only a small amount of publications are removed from a reference set. Small test cases are often algorithmically unresolvable as there is too little information available. Therefore, we provide sub sets of **case-based-all** which only provide large test cases. A detailed description of the data set can be found in Appendix C.1.

4.3.4 Creating an Embedded Test Collection

The case-based test collection provides a large number of examples of defects in DBLP. We will use this collection mainly to study properties of defects. However, the case-based collection is difficult to use for the evaluation of ER algorithms. (1) The test cases are isolated and provide only a limited context. Table 4.3 (Page 83) shows that some approaches require full coauthor network data, for example, to compute graph distances of reference sets. (2) There is little room for confusion. Classic test collections are set up so that algorithms are challenged. E.g., the collection of Han contains signatures from authors with similar names. While it is unclear how confusing this situation is to an algorithm that considers more than the surface form, the additional real-world entities create a potential for miscalculations. (3) The individual test cases are small. They cannot be used to evaluate the scalability of ER algorithms. In particular, it is not possible to evaluate blocking approaches. To solve these problems, we define a second test collection which integrates corrected defects in a full copy of the DBLP data. The embedded test collection consists of two parts.

- A full copy of the state of the DBLP collection at time t. Provided as a set of all records.
- Information on the defects which were present at time t and which were corrected later.

To obtain a significant number of annotated defects, we compare DBLP versions from different years. E.g., we compare the state of DBLP from the beginning of 2015 with the state at the beginning of 2017. A similar approach was presented by Reuther[Reu07]. Reuther considered two states of DBLP and compared signatures.

By searching for changing references to entities, Reuther determined merges and distributes which he used to create a test collection. We extend this approach in two ways:

- We distinguish between the three types of corrections: merge, split and distribute.
- Reuther limited the test collection to the relevant records. We will present a test collection which is embedded into a full version of the DBLP collection.

We apply the modification classification framework on the DBLP data with an observation framework consisting of just two time points. For the results, we apply the correction extraction and classification framework described in Section 4.2. The long interval between observations makes large deviations in surface forms more likely. In the fine-grained observation framework, we might have observed edits from *J. Doe* to *John Doe* to *John Doe-Miller*. In the framework used here, we might observe only a single edit from *J. Doe* to *John Doe-Miller*. Our metadata framework does not demand identifiers for individual data elements. Instead, it relies on value similarity and structural information to map data elements from different observations. The larger changes to surface form make this matching more complicated than for a dense observation framework. We expect more masked edits in general because of the long time between the observations.

We will also find more overlapping corrections than for the fine-grained observation framework. Assume that we observe a distribute operation between author profiles A and B (i.e., signatures are reassigned between A and B). Further assume that all signatures of a profile C are reassigned to profile A (a merge of C into A). For a sparse observation period as described above, there are many different ways in which these operations can be performed. In a slightly different situation, we might have observed a distribute between B and C and a merge of A into C. In particular, if all surface forms change, it is difficult to differentiate those cases. For the evaluation (See Section 4.3.5), we need clear merge and split cases. In cases as described above where distribute corrections overlap with splits and merges, we consider those operations as part of the distribute. I.e., in the example, we obtain a distribute $\{A, B, C\} \rightarrow \{A, B\}$ where signatures of $\{A, B, C\}$ are reassigned to $\{A, B\}$.

We applied the detection for different observation frameworks that each considered two states of DBLP at the beginning of different years. For this study, we use a historical framework which ends in February 2017. Table 4.5 lists the number of corrections we found for combinations of different dates. We do not consider states of DBLP from before 2013 as the collection was small at that time and the number of possible corrections is negligible. The number of corrections is small compared to the case-based collection (see Figure 4.5). The primary reasons are (1) short lived defects that were introduced to the collection and corrected between the observations are missing. (2) As discussed above, we might merge multiple corrections into one.

Table 4.5: Number of identified corrections for different observation frameworks, each consisting of two time points at the beginning of a year.

observation dates	split	merge	distribute	all
2013, 2017	2.207	19,175	5,346	26,728
2014, 2017	1.946	16,461	4,873	23,280
2015, 2017	1.536	13,393	3,968	18,897
2016, 2017	978	8,608	2,666	12,252

For each observation framework, we provide a list of corrections. Each correction consists of a set of source author profiles and target author profiles. The corrections reassigned signatures between these profiles. The profiles are identified by their DBLP person identifiers. We do not consider consistent renames. Example 4.4 shows a split correction with three target profiles. E.g., signatures from homepages/56/6789 were distributed to homepages/56/6789, homepages/56/6789-1 and homepages/56/6789-2. We also provide a second list which also contains the references assigned to the profiles before and after the correction.

Example 4.4: Split correction with three target profiles.

```
<defect type="Split">
  <source>
    <profile authorid="homepages/56/6789" />
  </source>
  <target>
    <profile authorid="homepages/56/6789" />
    <profile authorid="homepages/56/6789-2" />
    <profile authorid="homepages/56/6789-1" />
  </target>
</defect>
```

We created the embedded test collection for the observation frameworks listed in Table 4.5. Further details are described in Appendix C.2.

4.3.5 Application Scenarios for Test Collections

The test collections we presented in Sections 4.3.3 and 4.3.4 differ from those we discussed in Section 4.3.2 in two relevant ways:

- Our test collections do not directly provide correct examples. I.e., the collections contain defects, but no gold standard data for non-defective references.

- The corrections to defects can be partial. E.g., a homonym consists of three components, but a split extracts only one of them.

The collections described by Müller at al. [MRR17] (See Section 4.3.2) is optimized to evaluate batch algorithms. There are different ways to determine the quality of a batch ER algorithm. Santana et al. [SGLF15] describe two popular evaluation metrics. The K metric considers pureness and author fragmentation. An empirical reference set created by an ER algorithm is pure if it contains only signatures from the same theoretical reference set. A theoretical reference set is not fragmented if all of its signatures can be found in the same empirical reference set. The other metric is the *pairwise F1*. For this metric, pairs of signatures are considered. Let a be the number of signature pairs which the algorithms correctly assigned to the same empirical reference set. Let b be the number of signature pairs which are wrongly assigned to the same empirical reference set. And finally let c be the number of signature pairs which should have been assigned to the same empirical reference set but which are placed in a different empirical reference set. Santana et al. define pairwise precision as

$$P = \frac{a}{a+b} \tag{4.1}$$

and pairwise recall as

$$R = \frac{a}{a+c} \tag{4.2}$$

Their F1 metric is defined on top of this as

$$F1 = 2 \cdot \frac{P \cdot R}{P+R} \tag{4.3}$$

The combination of two measures is necessary to prevent trivial algorithms. Consider an algorithm which assigns each signature to a different empirical reference set. The empirical reference sets obtained by these algorithms have perfect pureness but maximal fragmentation. For classical test collections, a, b and c are provided by the ground truth. The test collections we present in this work provide this information only partially so we cannot compute precision and recall directly. The same is true for the K metric.

In this section, we discuss alternative evaluation strategies which use the test collections created for this work. We will mostly concentrate on the embedded test collection. The case-based collection will be used later to study properties of defects. The embedded collection provides three sets that describe the defective data:

- M: pairs of reference sets which are to be merged (the synonym defects).
- S: the reference sets which are to be split (the homonym defects).
- D: pairs of reference sets between which signatures are reassigned.

We use these sets to evaluate performance of ER algorithms.

Evaluation of Blocking Blocking is a preprocessing of the actual disambiguation algorithm. The goal is to create blocks which can be considered individually in later steps to save computation time. The main step of the algorithm processes all pairs of elements within a block, e.g., to cluster them into reference sets. There are two goals for blocking: elements which are to be grouped later should end up in the same block and the block size should be small. In Section 4.4, we will discuss how the case-based collection can be used to evaluate common blocking strategies. Here, we describe evaluation of blocking on the embedded DBLP collection.

The blocking component of ER algorithms relies strongly on recall. Recall for blocking is defined as the number of pairs of similar objects which are placed in the same block. For a batch-based algorithm, these are pairs of signatures from the same theoretical reference set. For a defect detection algorithm, these might be pairs of empirical reference sets that are synonyms. To satisfy recall, an algorithm could place all elements in the same block. However, we need to consider the size of the blocks as well. Evangelista et al. [ECdSJ10] propose an evaluation strategy for blocking that consists of *Coverage* and *Reduction ratio*. Let P be a set of all pairs of elements which are to be grouped together. A perfect blocking algorithm creates blocks in a way that for each pair in P both elements are in the same block. Evangelista et al. define *Coverage* as

$$C_{block} := \frac{|P^B|}{|P|} \tag{4.4}$$

where $P^B \subset P$ is the set of pairs which are placed in the same block. The *Reduction ratio* measures how many pairwise comparisons are needed after blocking. Let C be the number of all pairs of elements in the blocks. This is the upper bound of pairwise comparisons that the algorithm must compute. Let T be the number of all element pairs if the blocking returns a single block with all elements. Evangelista et al. define *Reduction ratio* as

$$R_{block} := 1 - \left(\frac{|C|}{|T|}\right) \tag{4.5}$$

The reduction ratio punishes algorithms that create large blocks. On et al. [OLKM05] use a similar coverage measure but consider size on a qualitative level when comparing blocking strategies.

We can apply a blocking algorithm to the data set of the embedded test collection. As the algorithm has to group all signatures, not just those affected by a correction, we can compute C and T – and therefore R_{block} – without limitations. P is unknown for the embedded test collection. However, a subset of P is available. With M as described above, we define a set of pairs of signatures as

$$P_{merge} := \bigcup_{m:=(A,B)\in M} \{\{a,b\} | a \in A \wedge b \in B\} \tag{4.6}$$

P_{merge} contains pairs of signatures which were originally placed in different reference sets but later merged together. As stated before, we can assume that the new reference set created by the correction has a high chance of being correct. We can define P^B_{merge} as the pairs of P_{merge} that are placed in the same block. C_{block} can be adjusted accordingly. Limiting the evaluation to P_{merge} instead of P removed most pairs of signatures in the collection from consideration. However, the effect is limited. (1) The embedded test collection provides many pairs of signatures with very different properties. (2) P_{merge} contains pairs of signatures which were placed in the wrong empirical reference set. In DBLP, this means that an error occurred during the initial processing. This indicates that these signatures are more difficult to place in the same reference set than other pairs of signatures. If a blocking algorithm can solve P_{merge}, it is likely that it performs well on the whole collection.

Evaluation based on partial data For the same reasons described for blocking, we cannot compute classical evaluation metrics for the embedded collection. However, we can approximate these measures to some degree. From an ER-algorithm we can expect two things:

- Signatures from profiles in M should be placed in the same empirical reference set.
- If a reference set is split, the signatures from the different components should not be mixed.

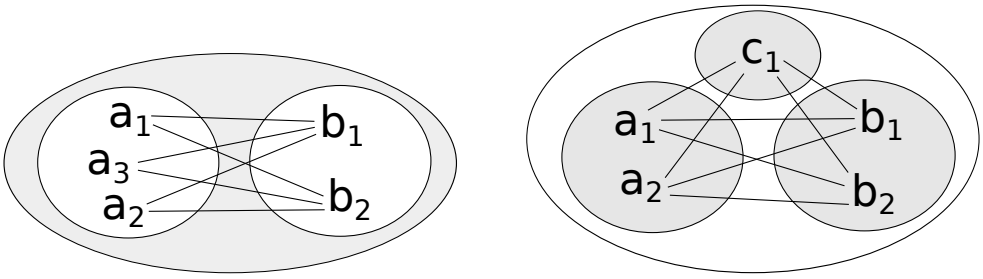
The evaluation scenario we propose is as follows:

1. Apply the ER-algorithm to the DBLP set in the embedded test collection.
2. From the reference sets created by the algorithm, remove all signatures which have not been part of a correction.
3. Use the remaining clusters for evaluation.

Step (1) ensures that the algorithm has access to all graph data. It also forces the algorithm to handle the data of a large collection which can be used to evaluate runtime performance. The limitation of step (2) ensures that only reasonably reliable data are used for evaluation. For the evaluation step we propose a measure based on pairs of references. Let P_{merge} as above. For each reference set $s \in S$, let $R(s)$ be the set of reference sets which are the result of the split of s. We define

$$P_{split} := \bigcup_{s \in S} \bigcup_{(s_1,s_2) \in R(s) \times R(s), s_1 \neq s_2} \{\{a, b\} | a \in s_1 \wedge b \in s_2\} \tag{4.7}$$

As for P_{merge} we only consider pairs of signatures from different reference sets. As before P_{split} contains pairs of signatures which where placed in different reference sets and are believed to be challenging. Also we must consider that split profiles might not be clean. Especially for split cases, we often observe that only a single component is extracted. An extreme case is the surface form *Wei Wang* from which



(a) P_{merge} pairs in a merge correction

(b) P_{split} pairs in a split with three components

Figure 4.6: Examples for merge and split pairs. Edges indicate pairs. Gray background: reference set after correction.

more than 60 components were split away during the observed time. Figure 4.6(b) shows an example of P_{split} with three target reference sets. We do not consider distribute corrections. Distribute corrections have a high chance of being partial corrections. I.e., a single defective signature is moved between the reference sets, but other defective signatures remain unchanged. We observed these situations frequently for the DBLP collection.

From the result of the algorithm, we determine four sets of signature pairs.

- P^+_{merge}, pairs from P_{merge} which are correctly placed in the same reference set.
- P^-_{merge}, pairs from P_{merge} which are wrongly placed in different reference sets.
- P^+_{split}, pairs from P_{split} which are correctly placed in different sets.
- P^-_{split}, pairs from P_{split} which are wrongly placed in the same reference set.

For these sets:

- $P^+_{merge} \cup P^-_{split}$ are the signature pairs placed in the same reference set.
- $P^-_{merge} \cup P^+_{split}$ are the signature pairs placed in different reference sets.

We can define precision and recall for merge and split pairs separately. For merge, we use split pairs to compute precision. For split, we use merge pairs for precision. We obtain

$$pre_{merge} = \frac{|P^+_{merge}|}{|P^+_{merge}| + |P^-_{split}|} \tag{4.8}$$

$$pre_{split} = \frac{|P^+_{split}|}{|P^-_{merge}| + |P^+_{split}|} \tag{4.9}$$

Recall is defined as follows:

$$rec_{merge} = \frac{|P^+_{merge}|}{|P^+_{merge}| + |P^-_{merge}|} \tag{4.10}$$

$$rec_{split} = \frac{|P^+_{split}|}{|P^+_{split}| + |P^-_{split}|} \tag{4.11}$$

From these measures, we can compute combined evaluation scores

$$pre = \frac{pre_{merge} + pre_{split}}{2} \tag{4.12}$$

and

$$rec = \frac{rec_{merge} + rec_{split}}{2} \tag{4.13}$$

The pair-based measures described above are more resilient against defective corrections than other evaluation measures such as cluster agreement. As pointed out above, corrections can be partial. Consider a reference set A with signatures $\{a_1, a_2, a_3\}$. Assume we see a correction that splits away another profile B so that we obtain $A : \{a_1, a_2\}$ and $B : \{a_3\}$. We obtain $P_{split} = \{(a_1, a_3), (a_2, a_3)\}$. Assume an ER-algorithm returns clusters $\{a_1\}$, $\{a_2\}$ and $\{a_3\}$. The result is consistent with our knowledge as we know that B was extracted from A but we cannot be sure if A was corrected completely (i.e., A remains a homonym). This situation frequently occurs in DBLP on which the embedded test collection is built. In the example above, cluster agreement between $\{\{a_1, a_2\}, \{a_3\}\}$ and $\{\{a_1\}, \{a_2\}, \{a_3\}\}$ is small but we obtain $P_{split} = P^+_{split}$. We can construct similar examples for incomplete corrections of synonyms.

P_{merge} and P_{split} are both significantly smaller than the set of all signature pairs in the collection. The combination of positive and negative edges defeats trivial algorithms but we might still observe undesired results from the algorithms. To mitigate the problem, we propose a combination of the evaluation described above with a classical collection.

- Compute the partial evaluation data for the embedded collection as described above.
- Compute classical precision and recall of the same algorithm in a classical test collection.
- Combine the results of both steps.

The results from the classical collection will ensure basic function of the algorithm and ensure comparability with results from the literature. Using the embedded test collection provides the following additional advantages:

- The scalability of the algorithm is fully tested.
- The algorithm is tested in test cases which might be underrepresented in the classical collection.

Failures in the embedded collection can also be used to qualitatively identify and fix weaknesses of the algorithm.

Use for training Many approaches which we described in Section 4.3.1 use supervised learning. A supervised learner must be supplied with examples. E.g., a supervised ER algorithm might be supplied with pairs of signatures that belong to the same author entity (positive examples) and pairs of signatures which belong to different entities (negative examples). The examples are used to train weights for similarity functions, neural networks and so on. While most learners are robust against occasional wrong input, a mostly correct collection of examples is required. Creating correct examples is the same problem as creating test collections. Many approaches use parts of test collections for training. E.g., Santana et al. [SGLF15] use 50% of the test collection for training. Both collections – case-based and embedded – can be used to provide examples for learners. In particular, the case-based collection provides a large set of realistic defect cases which the algorithm might use to identify hidden defects. For the embedded collection, P_{merge} and P_{split} can be used to create positive and negative examples for signature similarity.

The set of signatures from defects in the collection is mostly disjoint from classical test collections such as Han and Kisti because it is created in a different way. Using different signatures for training and evaluation is necessary to ensure correct evaluation results.

Scenario: Identifying problematic profiles Some ER approaches aim at detecting defects instead of recomputing the empirical reference sets. E.g., On et al. [OEL+06] describe a system which returns ranked pairs of synonym candidates. Santana et al. [SGLF15] incorporate a correction step that detects synonyms in their algorithm. We can assume that most real-world digital libraries have some approaches to identify defective reference sets. The tasks for defect detection are:

- Detect reference sets or sets of reference sets which are defective.
- Rank defects by likeliness of a defect.

Some approaches attempt to fix the defects automatically. DBLP uses algorithms which report suspect author profiles to a curator who will check them manually. As before, recall is straightforward for this scenario. For the embedded test collection, let M be the set of pairs of reference sets which are to be merged. Let $M^R \subset M$, $D^R \subset D$ be the pairs of reference sets reported by the defect detection algorithm. Let $S^R \subset S$ be the set of reported profiles for splitting. We can define recall as before. The problem is that the test collections cannot provide the set of all defects. Assume that an algorithm reports two reference sets as synonyms (suggesting a merge) that are not in the list of known defects. There are two possible scenarios: (1) The

algorithm returned a false positive. (2) The algorithm has uncovered a defects which has not been corrected. The possibility of case (2) prevents computation of a precision score. However, the embedded collection is still useful to determine if a significant number of defects can be detected. It can also be used to study undetected defects which might lead to the improvement of the ER algorithm.

4.4 Surface Form-Based Properties

In this section, we analyze similarities between surface forms of synonyms. Assume that an entity is referenced through two different and unmatched surface forms. While it is possible that the two surface forms are completely different, it is likely that they resemble each other in some way. For example, one surface form is an abbreviation of the other (e.g., *John Doe – J. Doe*). Many ER algorithms use surface-form based similarities as one feature to group signatures. Consider an algorithm that uses a blocking of signatures based on the surface form's last name. This might work well for collections where first names are often abbreviated and the last name is the more stable part. However, it is unclear if this blocking is transferable to other collections. In this section, we consider the surface forms of synonyms from the case-based DBLP collection described in Section 4.3.3. I.e., we consider surface forms of merge and distribute defects before the correction. We also use surface form pairs extracted from IMDB, Citeseer and Wikipedia. We first define a taxonomy for the similarity of person names, which we use to describe the different data collections. We show differences between the collections and discuss their possible implications. We also discuss the effectiveness of commonly used blocking strategies for different collections. In Section 4.5, we will consider network-based properties of all types of corrections.

In a first step, we classify pairs of synonymous surface forms based on the differences between them. The classification we use is derived from the work of Branting [Bra03] and Reuther [Reu07]. Branting determined nine categories of name variations, e.g., abbreviation of one name part. Branting's taxonomy is designed for organizational names. However, most aspects are also applicable to personal names. The classification of Reuther is an extension of Branting's, which also considers causes of synonyms such as marriage or divorce. The taxonomy we created is designed to classify pairs of synonyms automatically. To achieve this, we ignore causes of synonyms but concentrate on how they manifest. However, we will motivate the classes by discussing the underlying reasons. We defined seven main classes in which pairs of personal names can differ. For some classes we defined subclasses that provide a more detailed categorization. Table 4.6 lists the taxonomy with examples. Our taxonomy consists of the following classes:

Abbreviation: A full name part in one name is represented by an initial in the other name. In classical western publications, the first and middle names of authors are often abbreviated, but not the last name. In other cultures, abbreviations of the last name are possible. Abbreviations are common in scientific publications where

Table 4.6: Raw and fine categorization of name similarity for synonym surface form pairs.

Type	**Example**
Abbreviation (**A**)	P. Schefe ↔ Peter Schefe
Omission (**O**)	
▶ Token (**O-C**)	Lama ↔ Manuel Lama
▶ Suffix (**O-S**)	A. F. Tasch ↔ A. F. Tasch Jr.
▶ Prefix (**O-P**)	Dr. Michael Dell ↔ Michael Dell
Substitution (**S**)	John Doe ↔ John Miller
Order (**OR**)	Li Shujun ↔ Shujun Li
Processing (**P**)	
▶ Copy paste (**P-C**)	Uriel Jourdan ↔ Muriel Jourdan
▶ Typo (**P-T**)	Doanld Sofge ↔ Donald Sofge
Representation (**R**)	
▶ Diacritic, Ligature (**R-D**)	Luis Gouveia ↔ Luís Gouveia
▶ Upper/Lower case (**R-C**)	DaZheng Feng ↔ Dazheng Feng
Tokenization (**T**)	
▶ Whitespace (**T-W**)	Min Wook Kil ↔ Minwook Kil
▶ Other (**T-O**)	Zhi-Gang Tian ↔ Zhigang Tian

they are used to reference work in a compact way. McKay et al. [MSP10] point out other reasons such as masking gender or cultural background. Our analysis of the DBLP data set showed, that about 12.8% of all publications which were added between 2011 and 2015 were delivered with all first name parts abbreviated.

Omission: A part of one name is not part of the other name. Omissions occur because of space restrictions, unclear usage policies or cultural peculiarities. Again, this problem is quite common in scientific literature. E.g., Grossman and Ion [GI95] urged their colleagues to use a complete version of their names in all publications. Omission also covers cases where name parts are added, for example *Al-Hajji* might be added to the name of a person who completed a pilgrimage to Mecca. Names can be extended with prefixes and suffixes such as titles, degrees or generational terms such as junior or senior. Subclasses O-S and O-P refer to name additions like military ranks, post-nominal for orders or similar such as OBE or generational titles such as *junior* or *III*.

Substitutions: A name part is replaced by another, otherwise unrelated, word. Substitutions can be the result of marriage or divorce. Substitutions include usage of nicknames such as *Bill* for *William* or *Kees* for *Cornelis*.

Order of name parts is an important issue with cultures where the order of given and family name deviates from western tradition or where there is no fixed token order. Synonyms where name tokens appear in different orders can also be the result of unclear policies on how to record a name. E.g., when *John Doe* might also be provided as *Doe, John* for some documents.

Table 4.7: Comparison of name modification taxonomy with other taxonomies. ●: class fully present, ○: class partially present.

reference	A	O	S	OR	P	R	T
Branting [Bra03]	●	●		●	●	○(R-C)	●
Reuther [Reu07, pp 36-43]	●	●	●	●	●	●	●
Gong et al. [GWO09]	●	●		●			
On et al. [OLKM05]	●			●	○(P-T)		●
Qian et al. [QZS+15]	●	○(O-C)					●

Processing: Synonyms are created because of mistakes in the handling of names. This includes the classical typographical error. We also consider cases where the first or last letters of a name are missing. This often occurs when data are copied and pasted between different programs. We also considered OCR (Optical character recognition) errors such as *rn* merged to *m*. However, we found that this is not a relevant class of error for the data sets we analyze.

Representation: Synonyms might differ in the way they are presented in the data set. In this work, we concentrate on upper/lower case issues (R-C) and use of diacritics (R-D). We do not handle issues related to transcription of names as they are difficult to classify automatically. Using different cases for the otherwise same name might be a simple processing error. However, in some cultures, there is no fixed rule how to capitalize name parts, especially if the name is transcribed to the Latin alphabet.

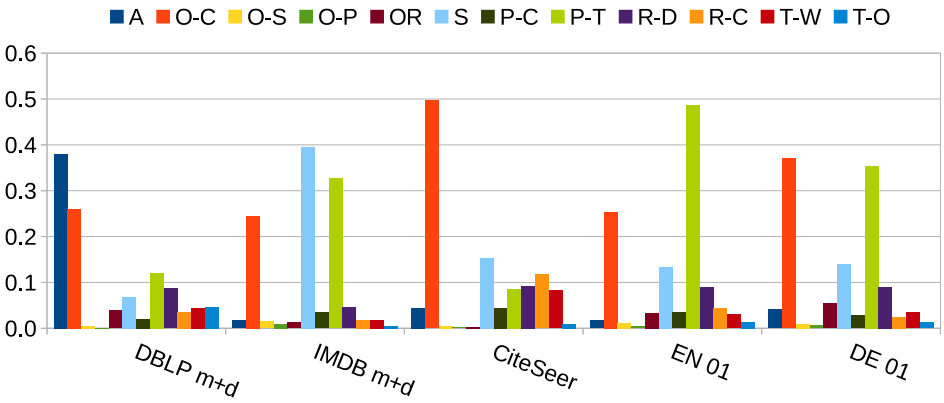
Tokenization issues occur when the number of tokens in a name is ambiguous. For example *Karl-Heinz* might be found as *Karl Heinz* or *Karlheinz*. We differentiate between tokenizations which change the number of words (T-W) and tokenizations which occur within the words (W-O). This is relevant as some string similarity approaches work on word level.

Table 4.7 compares our framework with frameworks developed by other groups. Gong et al. [GWO09], On et al. [OLKM05] and Qian et al. [QZS+15] all primarily deal with bibliographic data where abbreviations are common. Gong et al. concentrate on a small number of easy to detect modifications which they use as similarity measure. On et al. tried to create a realistic test collection with artificial errors. The categories in Table 4.7 represent what they consider to be relevant classes of problems. Class *R* is missing in these frameworks. We assume that most disambiguation algorithms normalize representation so it does not have to be considered.

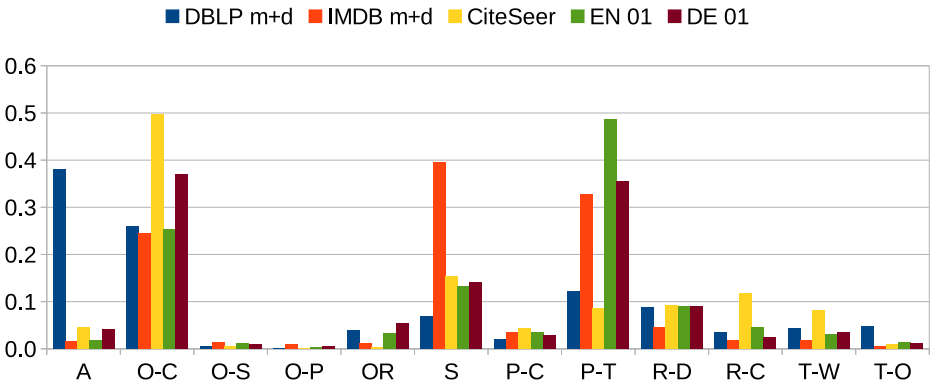
For our framework, we implemented an automatic classifier that determines the types of similarities between a pair of names. Multiple types can be assigned to the same pair. For example, the name pair *John A. Deo* and *John Adam Doe* would be classified as *A* (A. → Adam) and *P-T* (Deo → Doe). We classified name pairs for DBLP, IMDB, CiteSeer and Wikipedia data sets de_01 and en_01 (both contain personal data records). For DBLP and IMDB, we use the surface forms that are affected by merge and distribute corrections found in the data. For CiteSeer and

Table 4.8: Fraction of surface form pairs that were classified with a specific name similarity type. Pairs can be classified with multiple types.

project	A	O	OR	S	P	R	T
DBLP merge+dist	**0.380**	0.266	0.039	0.068	0.142	0.123	0.091
IMDB merge+dist	0.017	0.269	0.012	**0.396**	0.362	0.064	0.022
CiteSeer	0.045	**0.503**	0.003	0.153	0.130	0.211	0.092
E01	0.017	0.270	0.034	0.133	**0.522**	0.135	0.044
D01	0.041	**0.386**	0.054	0.140	0.383	0.115	0.047



(a) Comparison by project.



(b) Comparison by similarity type.

Figure 4.7: Distribution of name pair types in different projects. A name pair can have multiple types. See Table 4.8 for numeric results.

Wikipedia, we use edits related to personal names instead of classified changes. CiteSeer does not have a global time scope, so computing changes is not possible. The Wikipedia data sets are essentially authority records for person data. These records are isolated and all edits to them only affect a single record. I.e., the data there are not so much references to person entities, but standardizations of references. We do not expect homonyms or synonyms here. We cannot determine if edits corrected a synonym defect or not. However, the data we obtain for CiteSeer and Wikipedia give insights into the variability of name surface forms in those projects. For all projects, we classify name pairs according to the framework presented in Table 4.6 (Page 103). Some name pairs appear more than once, especially for distribute-type changes. In this case, we consider only one pair. For each project, we applied a name preprocessing to (1) remove project-specific markup in the names and (2) to normalize systematic differences in the representation of names. E.g., we changed *Doe, John* into *John Doe*.

Figure 4.7 shows the fraction of name pairs classified in a category. Table 4.8 lists the numerical results for the primary types. The data sets differ significantly. For DBLP, abbreviation (A) and adding of omitted tokens (O-C) are by far the most common operations. O-C is also an important class for IMDB, but most other pairs are classified as substitution (S) or processing type (P-T). We already discussed that abbreviated names are typical for scientific publications. Extending these abbreviated names greatly improved the quality of search results and is therefore a priority for DBLP [Ley09]. Apparently this is no problem for IMDB which has full first names for most persons. For both projects, a significant part of S is composed of name pairs where we can find the first name and a nickname version of that name. E.g., *Richard – Dick*, *William – Bill*. Using nicknames is common in both projects. We randomly selected 200 name pairs per project that were classified S. We marked pairs where one surface form contains a name and the other contains a fitting nickname. For IMDB, we found 71 nickname pairs and 51 pairs for DBLP. The other instances of S appear to be related to name changes after marriage or divorce.

CiteSeer and DBLP store records of the same type of documents. There is a significant overlap of the collections. However, abbreviation corrections rarely occur in CiteSeer. We need to keep in mind that CiteSeer groups synonyms to the same person entity by assigning a numeric identifier. Therefore, changes of surface forms are not necessary to merge groups of references. However, there is a significant number of modifications to data elements containing a personal name. We assume that most of these changes stem from improvements in the algorithms which extract the data from the documents. Most O type name pairs for CiteSeer omit an abbreviated name part. The reason is unclear but it might be an artifact of the data extraction software. Both Wikipedia data sets show a similar distribution of classifications. This is not surprising as we are dealing with the same type of data which is stored in a very similar form. However, there are measurable differences, especially for type P and type O.

The data show that the distribution of name pair type can vary significantly even for projects with similar content. We can only draw indirect conclusions on the

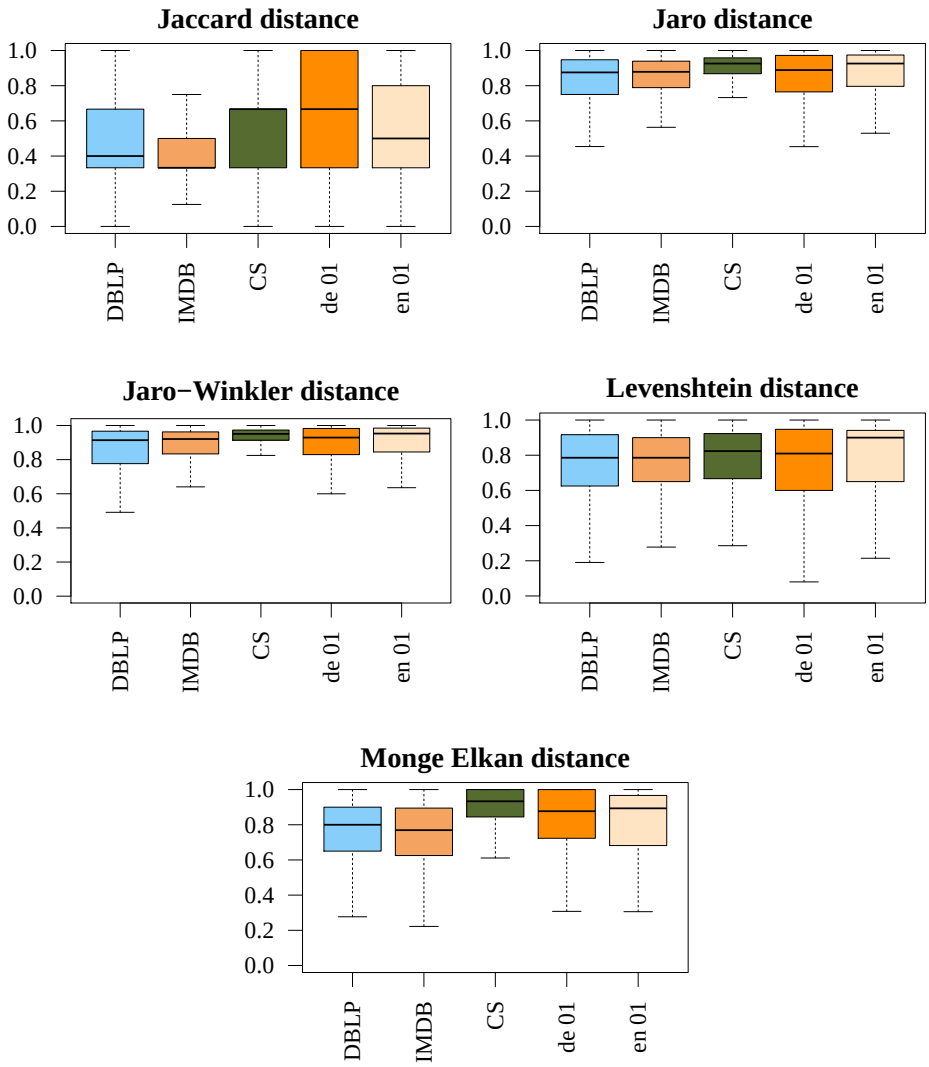


Figure 4.8: Results by Similarity Type. CS: CiteSeer.

properties of all synonyms. However, the numbers suggest that ER-algorithms might face different challenges in different collections. To get a basic estimate on the influence of the type distribution, we computed the similarity of name pairs using basic string similarity metrics. From the work of Cohen et al. [CRF03], we selected Jaro-Winkler, Jaro, Jaccard, Levenshtein and Monge-Elkan distances. We use scaled versions which produce similarities in $[0, 1]$. 1 represents a high match, 0 represents a poor match. Figure 4.8 shows the aggregated results for different projects. Keep in mind that changes and edits require a certain minimal string value similarity as described in Section 3.2.1. So for all name pairs, at least one of the similarities will return a good matching.

The results show that the performance of most algorithms varies depending on the project. One possible conclusion is that ER algorithms which use string similarities

Table 4.9: Comparison of abbreviation based similarity. The table shows percentage of pairs which are considered similar.

	consider case		ignore case	
project	**initial + last**	**last**	**initial + last**	**last**
DBLP merge+dist	76.51%	78.56%	77.10%	79.10%
IMDB merge+dist	46.24%	56.64%	47.15%	57.57%
CiteSeer	36.04%	43.32%	40.12%	47.73%
E01	46.06%	52.32%	47.53%	53.73%
D01	57.81%	65.72%	58.61%	66.33%

should use more than one function or adjust functions to the properties of their data. However, the data also show that the performance differences are small for most functions and it is unclear if the differences in performance are caused by the name pair type or by other factors.

We also considered another category of similarity functions that is primarily used for blocking. The similarities are based on the assumption that parts of a person's name are more stable than others. E.g., for western names, the last name might be more stable as it is usually not abbreviated or replaced by a nickname. For each name pair we extract the first and the last name. As this is difficult for general names, we use a heuristic which can also be found in the literature ([SGLF15], [HZG05]): The first name token is considered to be the first name, the last name token is considered to be the last name. All other tokens are ignored. The first token is than reduced to an initial, the last token is unchanged. E.g., *John Adam Doe* becomes *J. Doe*. Levin et al. [LKBJ12] use a variation where the middle initial is also considered if available. We then compute a binary similarity for those reduced names. If the reduced strings are identical, we return 1.0, else we return 0.0. We also implemented a version where the initial is omitted. This comparison method is similar to the approach used by Han et al. [HZG05] when they created their test collection. It is therefore an implicit blocking for all evaluations based on their approach. We also consider a version of each similarity where the case of the name is ignored.

The similarities described above are primarily used for blocking. In blocking, the goal is to assign surface forms that refer of the same entity to the same block. Table 4.9 shows the percentage of name pairs that are placed in the same block by a name based blocking approach. The blocking approaches we consider here are obviously designed with the name abbreviation problem in mind. The approaches perform well for DBLP with hit rates between 76.51% and 79.1%. This is due to the large number of abbreviation extensions in this project. The results for the other projects are significantly lower. While a hit rate of 0.791 is far from optional – of all name pairs 21% do not end in the same block – it might be acceptable. However, a hit rate of 46.24% will significantly reduce the usability of the algorithm. Santana et al. [SGLF15] do not use this similarity directly to compare synonyms but to consider the coauthors of potential synonyms. However, the potentially low hit

ratio is a problem as these coauthors might be synonyms as well. The results show that even basic algorithm components such as the blocking must be tested on the individual project. This is confirmed by the Wikipedia data sets that show quite different hit rates for similar data. The primary reason is the higher number of O-C name pairs in de_01 which in most cases concern the middle name and are therefore ignored.

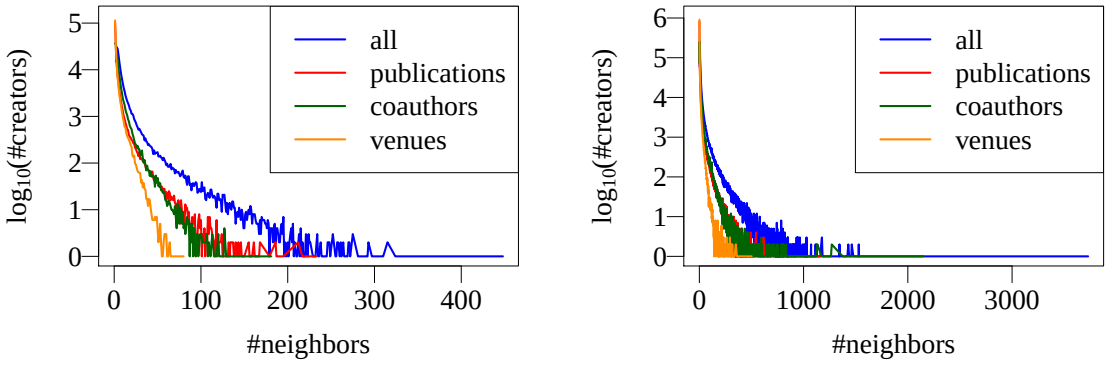
4.5 Graph-Based Properties

The case-based test collection (See Section 4.3.3) provides information about the empirical entities as local graphs. E.g., the coauthor relation is modeled as edges between nodes which represent person entities. We saw in Section 4.3.1 that local network information, in particular, the coauthors, is used by many state-of-the-art algorithms. In this Section, we will discuss the local graphs which we observed for the reference corrections. At first, we will describe the amount of information which is available from the graphs. In Section 4.5.2, we will discuss using the structure of the graphs as evidence for defects.

4.5.1 Locally Available Information

Some ER algorithms consider the similarity between reference sets, e.g., to find synonyms. E.g., Santana et al. [SGLF15] compute the similarity of reference sets based on sets of terms they extracted from publication titles and coauthor names. Approaches like this work best if a large amount of information is available to them. Assume that two reference sets are synonyms for a theoretical entity. If the sets are small, e.g., a single signature each, then the amount of data for comparison might be too small to detect the reference defect. If both sets provide a large amount of data, an algorithm is more likely to find information. In this section, we analyze the size of the neighborhoods of primary nodes in the test cases of the case-based test collection. The cases provide only local data. However, most ER algorithms use only local information.

Before we can discuss the available information per test case, we need to look at the available information in DBLP as a base line. The observation period for this study lasts from June 1999 till November 2015. During this time DBLP grew significantly. At this point, we will not look into details of the development of neighborhood size in DBLP during the observed time. Instead, we will show the underlying distribution and its stability over the time. We computed a metadata graph as described in Section 4.3.3 for the entire DBLP collection. At two points in time, we counted the number of publications, coauthors and venues in the direct neighborhood of each person entity. We also determined the total number of neighbors. Figure 4.9 shows the distribution of available information on January 1st 2002 and October 2015 (the end of the observation period). The x-axis shows the number of direct neighbors, the y-axis shows the number of authors that provide this amount of



(a) January 2002 (b) November 2015

Figure 4.9: Distribution of primary information sources in DBLP at two different times.

information. For both dates, we obtain an exponential distribution for all types of neighbors. This means that for a large number of author entities, we can expect a small neighborhood. We extracted data for all other years in the observation time frame and obtained similar results. We can safely assume that the amount of available information is exponentially distributed for the whole observed time.

We considered each primary node in the test collection, i.e., each empirical entity which is about to be split, merged or distributed. For each primary node, we counted the size of the neighborhood before the correction. The mean neighborhood size is 42.92, the median is 11. The mean number of publications is 14.78 (median 3), the mean number of coauthors is 20.17 (median 6) and the mean number of venues is 7.969 (median 2). The left part of Figure 4.10 shows boxplots for the distribution of these values. The neighborhood sizes are distributed exponentially which is consistent with the baseline of DBLP. These numbers do not tell the full story. Consider primary nodes a and b. Assume that the neighborhood size of a is smaller than the neighborhood size of b. If the size difference is large, an algorithm might not be able to compare a and b even if b provides a significant amount of information. From our work with DBLP, we know that homonyms and synonyms that involve multiple entities with large neighborhoods are rare. For each test case, we determined the primary entity with the largest (e_{max}) and smallest (e_{min}) neighborhood. For merge and distribute test cases, we compute these values before the correction. For split test cases, we consider min and max for results of the correction as well as the neighborhood size of the single primary node from before the correction. Figure 4.10 shows boxplots for the different sets of nodes, Figure 4.11 shows the distribution of the results. As expected, all size values we obtain are distributed exponentially. We can also see that the difference between e_{max} and e_{min} can be significant. For example, the mean number of coauthors of e_{max} in a merge correction is 21.4 while the mean size of the corresponding e_{min} is 3.3. In 4.6% of all merge cases e_{min} does not have any coauthor. 81.5% of e_{min} in merge cases

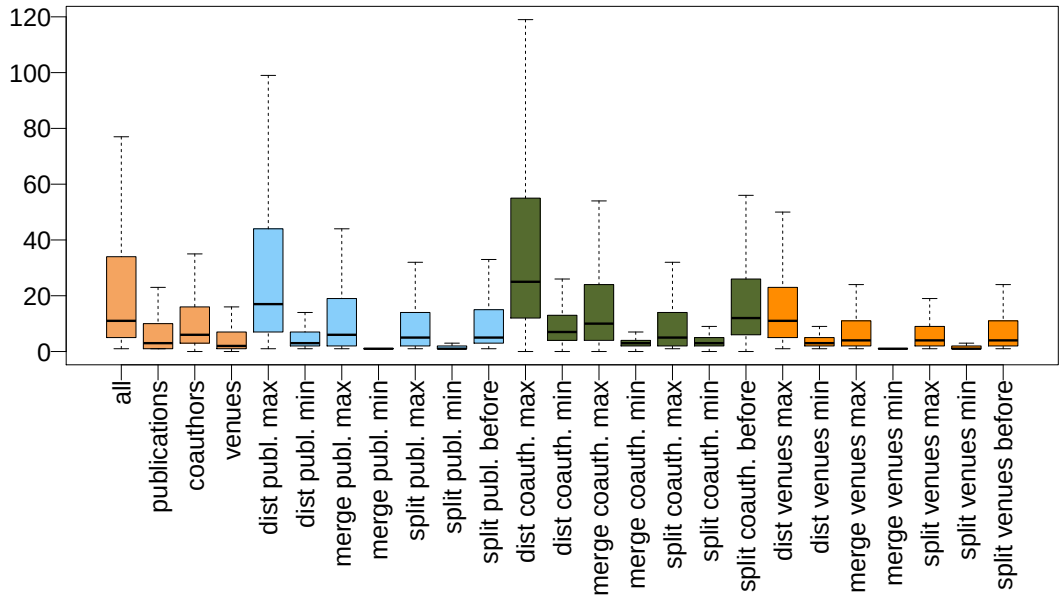


Figure 4.10: Boxplots for the distribution of neighborhood sizes. For merge and distribute, the min/max values are determined before the correction. For split, the min/max values are determined after the correction.

have less than 5 coauthors. Distribute type test cases provide more information for both e_{max} and e_{min}. This is probably because wrongly assigned publications are more often detected if both sides have much information available. Nevertheless, in 1% of all distribute test cases, e_{min} has no coauthors and in 30% of the cases, e_{min} has less than 5 coauthors. A small number of coauthors is problematic because coauthors are a frequently used network feature.

The results for publication and venue are similar. For merges, 83.9% of e_{min} are attached to a single publication. In almost all cases (98.6%) e_{min} has less than 5 publications attached. As the number of venues is bounded from above by the number of publications, the results here are similar. For split cases, we observe that often small author entities are extracted. For 53% of all split cases, the smaller entity is attached to a single publication only. I.e., a single publication was removed from an existing reference set. Detecting this single publication in a reference set can be algorithmically difficult.

The results on neighborhood size indicate that many test cases are algorithmically difficult because at least one of the entities involved is small. If an algorithm works on data from the collections alone (without linking external data), some test cases are unsolvable. To detect and remove these defects, manual effort might be required, for example, from authors who report defects to a project (see Section 5.2 for a study on user contribution to DBLP). It is difficult to estimate how much information is needed to apply an ER algorithm. For example, On et al. [OLKM05] create artificial synonyms where each empirical entity has 50 attached publications. While most algorithms will work well with less information, test cases where all primary

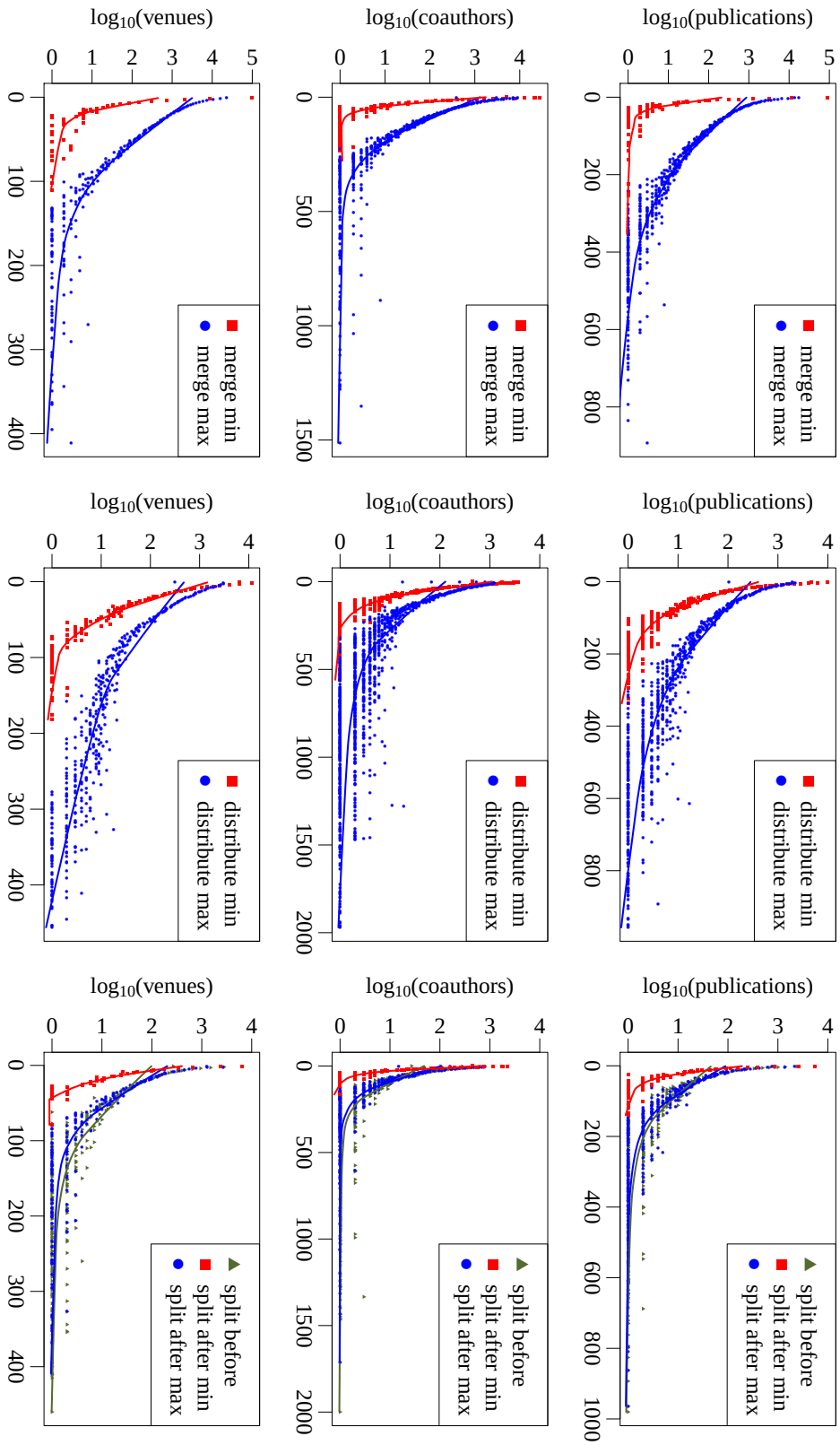


Figure 4.11: Distribution of the neighborhood size for primary nodes.

Table 4.10: Number of test cases where the smallest and largest neighborhood size of primary entities both have certain size.

		all			coauthors		
	total	≥ 5	≥ 10	≥ 50	≥ 5	≥ 10	≥ 50
merge	108,229	59,172	9,976	115	19,670	3,217	59
distribute	40,645	37,965	28,494	4,124	28,449	15,134	1,245
split	10,118	7,054	2,613	148	2,923	854	33

entities provide a certain amount of information are obviously simpler. For this reason, we created test collections which only contain cases where all primary nodes have a neighborhood size of at least 5, 10 or 50. Given the importance of the number of coauthors we also created test collections where the number of coauthors is at least 5, 10 or 50. Applying algorithms to these larger test cases might give a better insight into the algorithm's capabilities than running them on all test cases. Table 4.10 lists the number of test cases per collection. Note that the collections with neighborhood sizes of 50 or more almost completely consist of distribute cases. A detailed overview on the collection can be found in Appendix C.

4.5.2 Neighborhood Structure

After describing the amount of available data, we will now consider the structure of the entities' neighborhoods more closely. One application is the detection of pseudo entities which represent multiple persons (homonyms). The fact that an entity represents multiple persons can manifest in the neighborhood structure. An algorithm can detect these patterns.

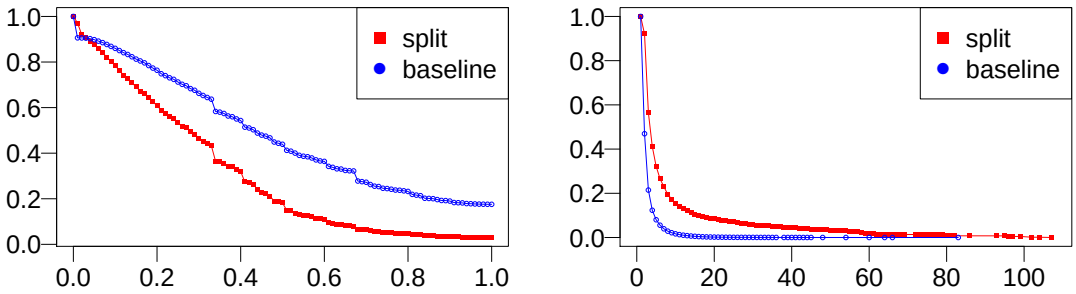
Let n be a node and let $G_n := (V_n, E_n)$ be a part of the metadata graph which contains the coauthor neighborhood of n. That is (1) all nodes which are connected to n by a co-created (coauthor) relation (2) all co-created edges between these nodes. We extract G_n for all primary entities in split test cases before the correction. For these G_n, we compute the local undirected clustering coefficient [WS98]

$$cc_n = \frac{2 \cdot |\{e \in E_n\}|}{|V_n| \cdot (|V_n| - 1)} \tag{4.14}$$

The local clustering coefficient describes the probability that two neighbors of n are connected. If $cc_n = 0$, the coauthors of n never cooperated with each other. $cc_n = 1$ means that G_n is a clique. We ignore split cases where n does not have any coauthors. Entities which are about to be split represent multiple real world persons. Normally, these persons are unrelated (except for a common name) and therefore do not share coauthors. We expect cc_n to be low for these entities as – in most cases – there are no edges between different coauthor groups. Clean author entities on the other hand often cooperate with their coauthors for a long time and are expected

Table 4.11: Local network properties for split nodes and baseline. f_0, f_1: percentage of nodes where value is 0 or 1 respectively. $1.Q$, $3.Q$: first and third quantile.

		mean	**median**	$1.Q$	$3.Q$	f_0	f_1	**min**	**max**
cc_n	split	0.304	0.274	0.115	0.429	2.9%	3.0%	0	1
	baseline	0.518	0.467	0.267	0.800	9.4%	17.6%	0	1
col_n	split	7.54	3	2	6	0.2%	7.8%	0	127
	baseline	2.123	1	1	2	2.3%	51.9%	0	83



(a) $p(cc_N \geq x)$ (b) $p(col_n \geq x)$

Figure 4.12: Aggregated clustering coefficient and color count for split cases compared with baseline.

to have a high clustering coefficient. As baseline, we randomly selected 10% of all reference sets found in DBLP in October 2015 which have more than one signature. Reference sets with a single signature cannot be a split candidate. The majority of the selected reference sets do not represent multiple real world persons. Table 4.11 shows statistic properties of cc_n. Figure 4.12(a) shows the aggregated distribution of cc_n for the baseline compared with the results we obtained for split candidates. As expected, cc_n for the baseline is higher than cc_n for test cases. However, for both split candidates and baseline, we find a relevant number of nodes with $cc_n = 1$ and $cc_n = 0$. We considered split candidates with $cc_n = 1$. There are three possible reasons: (1) the real world authors behind n actually worked with the same person (2) one of the real world authors does not have coauthors and (3) one (or more) of the coauthors are homonyms themselves and thereby create defective connected components. This case is common for authors with Chinese names in DBLP.

A clique might not be a realistic expectation for the coauthor graph of a researcher. Consider a professor P who works with students A and B and some time later with B and C. Though A and C did not collaborate directly, they belong to the same community. DBLP uses a simple clustering algorithm to capture these situations. Let G_n be as above but with n and all its edges removed. We compute the connected components, i.e., two nodes are placed in the same component if there is a path between them. DBLP assigns colors to each component and displays them

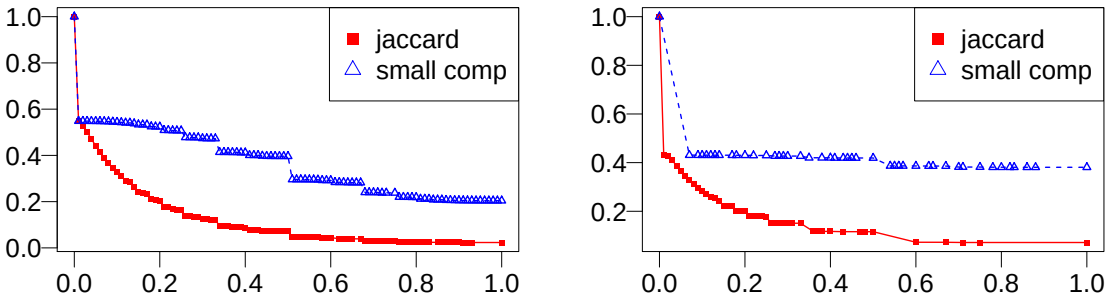
on the project's web site. Let col_n be the number of connected components in the neighborhood of node n. For a *clean* person node, we expect col_n to be small. For a split candidate, we expect to observe additional connected components from the different authors it represents. Figure 4.12(b) shows the aggregated distribution of col_n for splits and the same baseline as described above. Table 4.11 shows statistic properties of col_n. col_n for split candidates is larger than col_n for the baseline. The effect is more pronounced than for cc_n, however, we must consider that DBLP uses a variation of col_n to find homonyms already. So split cases with a high color count are overrepresented in the test collection. While only 7.8% of the split candidates have a single connected group, 51.9% of baseline nodes have this property. However, there is a number of nodes in the baseline set with high col_n values. While these might be hidden homonyms, having multiple connected components is possible without a defect involved. The split candidates with very high col_n values are caused by two large homonyms *Wei Wang* and *Lei Zhang* that are not in the baseline.

The relative small differences between split candidates and baseline nodes show that cc_n and col_n alone are insufficient predictors for homonyms. However, combined with other features, both measures could be used to improve homonym detectors. cc_n and col_n could also be used in a prescreening step to determine possible candidates which will than become subject to a more detailed and possibly more expensive analysis.

We will now analyze how neighborhoods of different nodes are related. Related neighborhoods are used to detect synonyms who worked with the same coauthors (e.g. [FVGL10][HGZ+04][LH10][SGLF15]), published on the same venues (e.g. [FVGL10]) and worked on similar topics (e.g. [HZG05]). We will also consider the results of split corrections to see if the resulting nodes are still connected. Let R be a set of relation types, such as *co-created*. Let V_n^R be the neighborhood of n within the graph defined by R. Let n_1 and n_2 be nodes and V_1^R and V_2^R their neighborhoods for some R. We consider three different functions to compare V_1^R and V_2^R: $count(n_1, n_2)$ counts the number of common nodes. $jac(n_1, n_2)$ computes the classical Jaccard distance between the sets. In the previous section, we saw that entities involved in a correction often have different neighborhood sizes. The Jaccard distance implicitly assumes that both sets are of the same size. Assume that V_1^R is smaller than V_2^R but $V_1^R \subset V_2^R$. This is a strong indicator that n_1 and n_2 are synonyms, however, the $jac(n_1, n_2)$ will be bound from above by the size of the smaller neighborhood. To take this effect into account we also compute a measure based on the smaller component. Let $|V_{min}^R| = \min\left(|V_1^R|, |V_2^R|\right)$. We define the *small comp distance* as:

$$sn(n_1, n_2) := \begin{cases} \dfrac{|V_1^R \cap V_2^R|}{|V_{min}^R|}, & \text{if } |V_{min}^R| > 0 \\ 0, & \text{else} \end{cases} \tag{4.15}$$

We compute all three values for $R^C = \{\text{co\_created}, \text{co\_contributed}\}$ (the coauthors) and $R^V = \{\text{created\_at}, \text{contributed\_at}\}$ (the venues the author published on).



(a) overlap coauthors

(b) overlap venues

Figure 4.13: Aggregated overlap of coauthors and venues for merge correction cases. Y-axis shows $p(value \geq x)$.

The values we obtained are an approximation for the actual overlaps. This is because we need to rely on the empirical reference sets in DBLP instead of using the theoretical reference sets. Assume a synonym defect with reference sets a_1 and a_2 referring to the same real world author A. Further assume that publications from both reference sets were coauthored with author entity B. If B itself is referenced by a synonym of profiles b_1 and b_2, it is possible that we obtain coauthor edges between reference sets a_1 an b_1 as well as a_2 and b_2 but no common coauthor of a_1 and a_2. A similar situation can occur when two reference sets are connected by a homonym without actually sharing coauthors. Figure 4.13 shows the distribution of jac and sn. Table 4.12 shows the statistical properties of all three values. 46.4% of the merge candidates do not share a coauthor and 56.9% do not share a venue. 29.9% are not not share coauthor or venue. This is not surprising as many components of a merge are small and the chances for an overlap are not high. Small overlap is problematic as many ER-algorithms rely on shared information in the neighborhood. I.e., for some ER-algorithms, these test cases would be hard to solve. Nevertheless, intersections for merge test cases are more common than intersections between random reference sets. From the baseline set described above, we randomly selected 2000 nodes and compared them pairwise. We found that only 795 (0.02%) of the pairs share a coauthor and 50,684 (1.27%) pairs share a venue. jac and sn are small accordingly. We also note relatively large f_1 scores for sn. This indicates that even if components of a merge are small, there is some chance that an overlap can be found.

We also considered split results after the correction. We expect that the reference sets have no significant shared neighborhood after the split. 6.83% of all split result pairs share at least one coauthor and 13.5% share at least one venue. The intersection of venues is not surprising as it is quite possible that several authors with a similar name publish on the same conference or journal. There are three explanations for pairs with shared coauthors: (1) The authors actually worked with a common coauthor. This might be the reason why the references ended up in the same reference set. (2) One of the coauthors is a homonym and needs to be

Table 4.12: Overlap between neighborhoods. See Table 4.11 for explanation of symbols.

		mean	**median**	$1.Q$	$3.Q$	f_0	f_1	**min**	**max**
R^C merge	*count*	1.120	1.000	0.000	2.000	46.4%	n.a.	0	84
	jac	0.117	0.025	0.000	0.143	46.4%	2.3%	0	1
	sn	0.360	0.250	0.000	0.667	46.4%	20.4%	0	1
R^V merge	*count*	0.459	0.000	0.000	1.000	56.9%	n.a.	0	14
	jac	0.136	0.000	0.000	0.125	56.9%	7.2%	0	1
	sn	0.405	0.000	0.000	1.000	56.9%	38.1%	0	1
R^{C+V} merge	*count*	1.579	1.000	0.000	2.000	29.9%	n.a.	0	85
	jac	0.104	0.048	0.000	0.143	29.9%	0.6%	0	1
	sn	0.378	0.333	0.000	0.667	29.9%	11.5%	0	1
R^C split	*count*	0.154	0.000	0.000	0.000	93.17%	n.a.	0	53
	jac	0.013	0.000	0.000	0.000	93.17%	0.02%	0	1
	sn	0.036	0.000	0.000	0.000	93.17%	1.58%	0	1
R^V split	*count*	0.193	0.000	0.000	0.000	86.5%	n.a.	0	32
	jac	0.029	0.000	0.000	0.000	86.5%	0.9%	0	1
	sn	0.082	0.000	0.000	0.000	86.5%	5.5%	0	1

split as well. (3) The split is incomplete and more publications need to be reassigned. While (1) is not likely, there are several examples of this constellation in DBLP. (1) and (2) pose problems for a defect correction algorithm as it needs to handle the link between the authors. Having 53 common coauthors is an outlier case of scenario (3). It was created when *Y. Suzuki* was split into *Y. Suzuki* and *Yoshinao Suzuki*. During this operation, a single paper with more than 53 authors was not reassigned. The paper was reassigned to *Yoshinao Suzuki* on the following day. This operation has been detected as a distribute.

4.6 Prediction of Hidden and Future Defects

In this section, we present an exploratory study on how to use corrections to predict undiscovered data defects. The work presented here has been published in [RH13] which is an extended version of a paper published in ASONAM 2010 [RH10b]. It is joint work with Oliver Hoffmann who implemented the community extraction described below. The remaining excerpts of this work presented here are sole work of the author of this thesis.

This study is inspired by the field of software engineering where bug or failure prediction based on properties of the source code is an active field of study [SLM10][FAW08]. In the simplest case, the probability of a bug in a software artifact (a package, class, method ...) is estimated by fixes to previous bugs in that component. The idea is that many defects are not isolated but caused by properties

of the component such as code complexity or mutability. Artifacts with many fixes in the past are considered to be more likely to need fixing in the future. In this section, we apply the same strategy to corrections in DBLP. The central artifact in DBLP is the author profile. We saw in the previous sections that signatures can be assigned to the wrong author profile. A correction takes away a publication from one author profile and assigns it to another one. We group the author profiles in DBLP into different communities. For each community, we count the number of corrections to record assignments (removing record or adding record) and test how this number can be used to predict the number of future defects. We will show that:

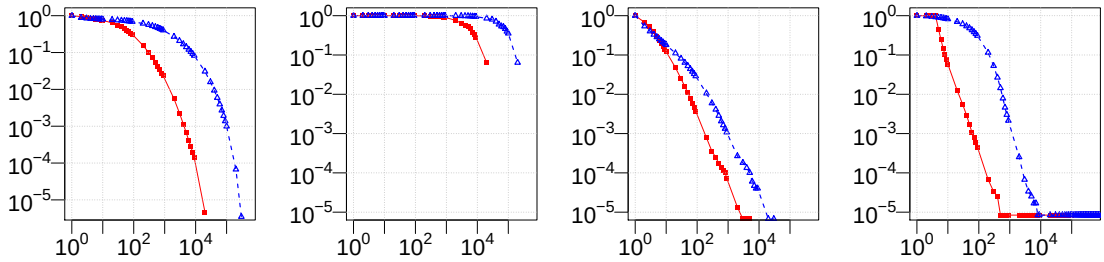
- Some author communities are affected by more corrections than others.
- There are tentative results that defect distribution can be used to predict hidden and future defects.

This study was performed with a preliminary version of the change extraction framework. It covers data observed between September 1999 and October 2010. As stated above, we will only consider corrections of person reference defects, i.e., defects where a publication was assigned to the wrong author.

4.6.1 Community Extraction and Reliability Estimation

We saw in Section 4.5 that most author profiles have few signatures. This means that for many author profiles, we will not find a correction in the observed time. To get an estimation for all profiles, we group them to communities based on their coauthors and their publication venues. The factors that influence the data quality and thereby the distribution of corrections are unknown. In this work, we follow the anecdotal knowledge that correction denseness is often related to author communities. E.g., a community of Chinese authors will see many corrections that are caused by the high ambiguity of Chinese names. In other communities, abbreviated first names are still common (e.g. electrical engineering). Name abbreviations are a central cause for homonym and synonym defects in DBLP.

We use three different community extraction strategies. For each strategy, we extract a set of (possibly overlapping) communities for the whole DBLP collection. The first strategy considers the neighborhood of authors and venues. For each author profile in the data set, we define a community consisting of this profile and the direct coauthors. We refer to this set of communities as A_C. For each venue in DBLP, we define a community as the authors who published there, i.e., the author profiles where at least one signature is from this venue. We refer to this set of communities as A_V. The second strategy uses an algorithm introduced by Radicchi [RCC+04]. It computes closed paths to determine whether an edge runs inside a community or connects two of them. Successively, edges between communities are removed till disjoint sets of names remain. The results are expected to be similar to the results of the algorithm by Girvan and Newman [GN02] for most graphs but the algorithm



(a) A_C (b) A_V (c) R (d) P

Figure 4.14: $p(size > x)$ for different set of communities. Red squares: size in number of author profiles per community, blue triangles: size in number of signatures per community.

Source: Adapted from [RH13]

is much faster. We use a weighted version of Radicchi with a path length of 3 on the coauthor graph of DBLP. The algorithm can compute weak and strong communities. Communities created with the strong approach are very large. To allow a fine grain computation, we use the weak communities approach. We refer to these communities as R. Communities in R are not overlapping. For many data sets, this is not realistic as some authors are part of two or more distinct communities. To obtain overlapping communities, we apply the Clique Percolation method by Palla et al. [PDFV05]. Communities are defined as unions of k-cliques which share at least $k-1$ nodes. For this study we use $k = 4$ as $k = 3$ created large communities (close to the giant component) and $k = 5$ requires significant computation time. The approach ignores edge weights. Clique Percolation ignores all authors which are not part of at least one k-clique, for example, authors without coauthors. We apply Clique Percolation to the DBLP coauthor graph. We refer to the set of communities extracted with this algorithm as P. Figure 4.14 shows the distribution of size (author profiles and signatures) for the different sets of communities. A_V contains by far the largest communities. The distributions of size for A_C, R and P are similar.

Let c be a change that reassigns signatures from author profile a to author profile b. We count this as a correction for the community of a and the community of b. Reassignments within the same community are counted twice as there were two defective author profiles (one missing publications, one with too many publications). We determine the number of corrections for each community in all community sets. Let $|\mathcal{C}|_A$ be the number of author profiles in $\mathcal{C}$ and let $|\mathcal{C}|_S$ be the number of signatures in community $\mathcal{C}$.

We compute three correction density values:

- The *raw frequency* $rf(\mathcal{C})$ is the number of all corrections to $\mathcal{C}$.
- The *author normalized frequency* defined as $af(\mathcal{C}) := \frac{rf(\mathcal{C})}{|\mathcal{C}|_A}$.
- The *signature normalized frequency* defined as $sf(\mathcal{C}) := \frac{rf(\mathcal{C})}{|\mathcal{C}|_S}$.

For each frequency measure and each type of community, we compute a risk score for the authors it contains. Let a be an author profile, let $C \in \{A_C, A_V, R, P\}$ be a set of communities and let $f \in \{rf, af, sf\}$ be a frequency measure. Let $\mathcal{C}_1, \ldots, \mathcal{C}_k \in C$ be communities which contain a. We define the reliability of an author profile a as

$$r_{f,C}(a) = \max_{1 \leq s \leq k} f(\mathcal{C}_s)$$

A high risk score indicates that this author profile is prone to corrections.

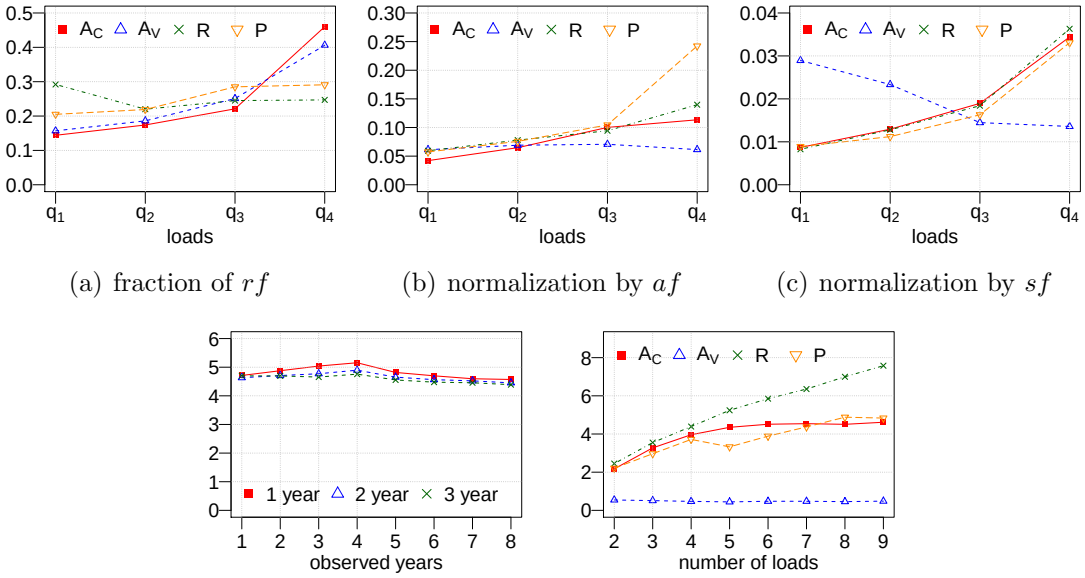
4.6.2 Prediction Capabilities for DBLP

In this section, we analyze if the risk estimation for author profiles can be used to predict future corrections.

Experiment

We split the observed time frame in two parts, June 1999 – October 2007 and October 2007 – October 2010. We extract communities as described above from the end of October 2007. For these communities, we compute the risk scores, using corrections from different periods of time. I.e., we compute r using the corrections from one year before 2007, two years and so on. The second part of the observed time frame is used for evaluation.

For each community set C and frequency measure f, we partition the author profiles in bins $q_1, \ldots, q_k$ according to their risk score. q_1 contains the author profiles with the lowest $r_{C,f}(a)$ while q_k contains those with the highest scores. The bins have the same size with respect to the number of publications assigned to the author profiles. We vary k between four and ten. For the evaluation, we count the number of corrections after October 2007. We normalize those values in the same way we normalized the results from the observation period. We consider evaluation periods of one, two and three years. We compute rf, af and sf considering only the corrections from the evaluation period. If our assumptions are true the average correction densities in q_1 will be smaller than the correction densities in q_k. Many communities with no corrections are very young and usually small. Our experiments show that with this lack of data it is difficult to compute a reliable correction prediction. To get usable



(a) fraction of rf (b) normalization by af (c) normalization by sf

(d) influence of time settings (e) influence of number of bins

Figure 4.15: Prediction results for different settings.

Source: Adapted from [RH13]

results, we ignore profiles a with $r_{\cdot,\cdot}(a) = 0$. We only apply the estimation for author profiles which were in the DLBP in October 2007. During the evaluation period, structures within the networks might change in a way that we obtain different communities at the end of the evaluation period. Evolutionary clustering [CKT06] might solve this problem provided that reasonably fast algorithms become available.

Results

In a first step, we consider four bins. Figures 4.15(a), 4.15(b) and 4.15(c) show the average normalized correction density for each community in the different bins. For A_C and P, we observe that q_4 has a higher normalized correction density than q_1. R shows the same pattern for af and sf. For normalization sf the defect density in q_4 is between 3.5 and 4.4 times higher than the density in q_1. A_V does not produce a good prediction of corrections. This was expected as (1) the communities in A_V are very large and will not capture effects on data quality well. (2) many known effects relate to coauthor communities and not to venue communities (origin of names, handling of own name when publishing ...).

Observation and evaluation time have only a small influence on the results. Figure 4.15(d) shows the quotient q_4/q_1, i.e., the factor by which the correction density increases for an evaluation period of one, two and three years and an observation period between one and eight years. Again, we used Communities P and normalization sf which produced the best results. The small influence of the years used

for prediction can be explained by the fact that most records in DBLP are not old enough to be corrected several years ago. It is also reasonable to assume that communities with many corrections recently will be corrected again in the near future. There is a similar explanation for the small influence of the evaluation period. We consider corrections only for authors for whom we have a prediction. This excludes all authors added after the beginning of the evaluation period. With an increasing length of the evaluation period, more and more corrections are not considered. Finally, we tested the influence of the number of bins. Figure 4.15(e) shows the changes to q_k/q_1 for different numbers of bins. We use sf normalization. Again A_V fails to predict corrections but R profits from distinguishing between more bins. For this method, the density of corrections in q_9 is 7.58 times higher than the density for q_1.

The most obvious factor in assignment quality is the name itself. Many communities with low reliability consist of entities where names are abbreviated. The mean percentage of author profiles with abbreviated names in reliable communities is 6.4%. In unreliable communities 44.6% of the names are abbreviated. Even if a name is given in full there are differences in the ambiguity as already noted by Bhattacharya et al. [BG07]. We randomly examined selected communities and found a large number of names which are of Chinese origin. Many different Chinese names map to the same transcription in the Latin alphabet. This introduces ambiguity which favors homonyms.

Conclusion and possible Application

The experiments show that prediction of corrections for individual authors or publications is not possible. However, it is possible to determine communities which will most likely be affected by corrections in the near future. A possible application for this knowledge is to annotate affected author profiles with a warning about the data quality. This might be beneficial to the user of the DBLP collection.

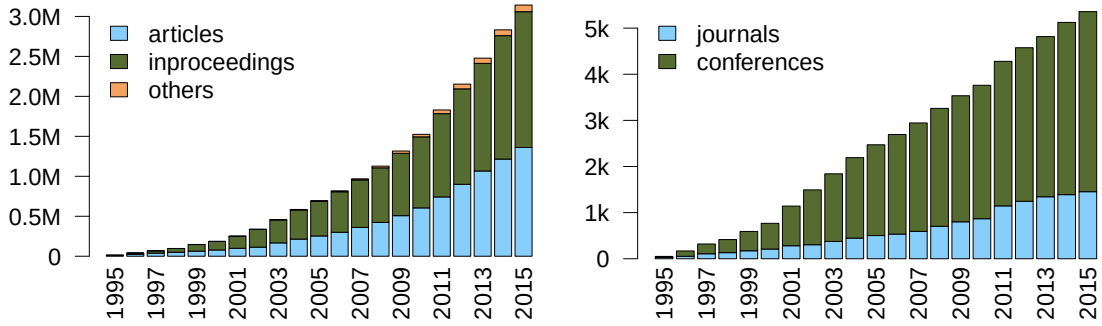
Chapter 5

Process Analysis

Contents

Besides detecting defects, we can use information on a collection's past for a variety of studies on its history. Many digital library projects were established years ago and provide sufficient information to study their development. This is not only of historical interest. Past development can be used to understand properties of the collections and allow predictions of future development. Past information is also helpful to understand the impacts of procedural changes in the past. For example, we can determine whether a new data acquisition policy increased the number of new records or changed the data quality.

In this chapter, we present two studies based on the historical metadata of the DBLP project. In Section 5.1, we discuss several aspects of the evolution of the collection. In particular, we study the thematic development of the collection. In Section 5.2, we show how historical metadata can be used to estimate the influence of users on a collection. Both studies were published previously as [RH10a] (Section 5.1) and [RH11] (Section 5.2). Both publications are coauthored by Oliver Hoffmann. The excerpts from those publications presented in this thesis are sole work of Florian Reitz.



(a) Number of records in millions

(b) Conferences and Journals

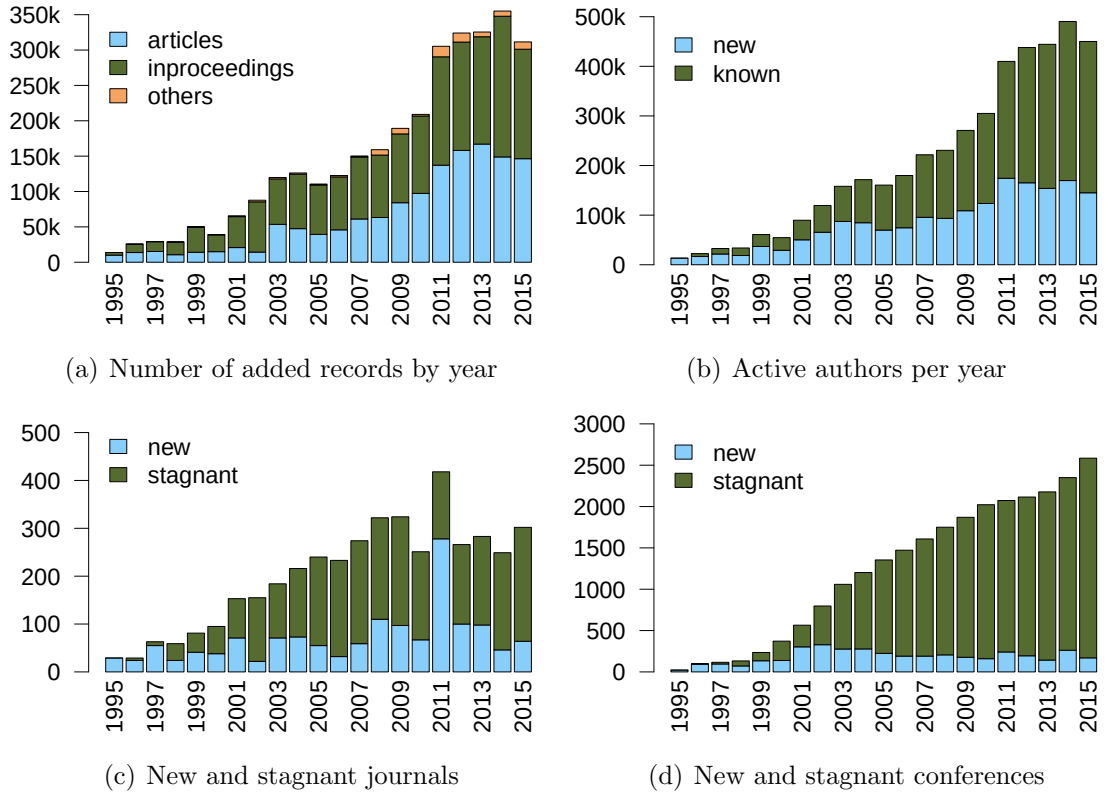
Figure 5.1: Aspects of the evolution of DBLP between 1995 and October 2015. The Figures show the state of the collection at the end of a year (for 2015, the figure shows the state at the end of October).

5.1 Evolving Coverage in DBLP

Since the 1990s, the DBLP collection has been subject of various studies on the field of computer science. For example, DBLP has been used to study collaboration and publication patterns of the field (e.g. [MCM15][DKL08] on expert detection, [HYQQ09][EL05] on community structure and publications). Despite its importance, little work has been done to analyze the collection itself and its development. Not only did the collection grow significantly, there are also changes to the thematic composition. The thematic change best reflects in the change of name. Originally, DBLP stood for *database systems and logic programming* suggesting a strong thematic focus. The name was later changed to *Digital Bibliography & Library Project*. Understanding these changes is important to judge how representative DBLP was of computer science at different times. This can influence the comparability of studies which used different versions of the data set. In the following sections, we will first discuss the growth of DBLP in general. In the next step, we will show how thematic changes of a collection can be reconstructed from historical metadata.

5.1.1 Growth of DBLP

Before we can discuss the topical development of DBLP, we need to consider the growth of the collection in general. For this section, we use an observation framework that starts in October 1995 and ends in October 2015. As stated in Section 2.4, modifications of DBLP can be observed reliably since June 1999. However, we can determine the record lifetime since October 1995. I.e., we can track the day a record was added to DBLP. For the study we resent here, reconstructing all modifications is not necessary. Figure 5.1(a) shows the number of publications stored in DBLP at the end of a year (for 2015, the Figure shows the number of publications at the end of October, the end of the observation period). The dominant types are articles from journals and conference proceedings. The other publications are books, collections



(a) Number of added records by year

(b) Active authors per year

(c) New and stagnant journals

(d) New and stagnant conferences

Figure 5.2: Different indicators for the growth of DBLP between 1995 and 2015. Figures for 2015 are incomplete as the observed timeframe ended in October.

and theses. Figure 5.1(b) shows the number of different journals and conferences which were listed in the collection.

Figure 5.2 gives more details on how these numbers developed during the observed timeframe. Figure 5.2(a) shows the number of added records per year. The significant increase between 2010 and 2011 is the result of an increased funding of the project. In the previous chapter, we saw that the majority of authors in DBLP only published one or two papers. To maintain the number of new publications in the collection (assuming a stable number of authors per paper), new authors must be added frequently. Figure 5.2(b) shows that recently more than 150,000 new authors were added each year. The fraction on new authors on the number of all active authors in a year is decreasing. Between 1995 and 2003 more than 50% of the active authors were not known before. For 2013 to 2015 the share of new authors was 34.6%, 36.7% and 32.2% respectively. However, the growth is significant and community structures might change in relative short time.

Figures 5.2(c) and 5.2(d) show the number of journals and conferences by year. A venue is considered stagnant if no new publication was added to it in that year. A venue is new if it appears for the first time. We omitted known venues with new publications in both figures. While many journals are alive (neither new nor stagnant),

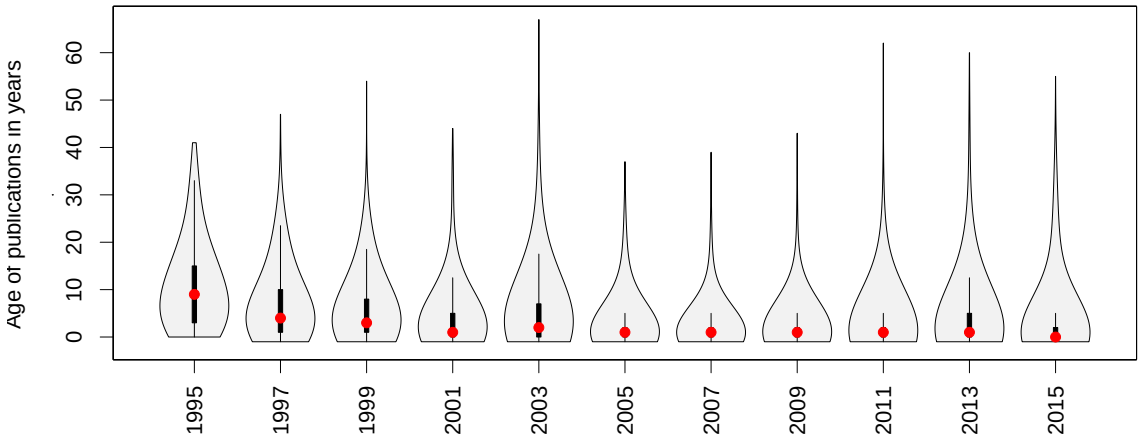


Figure 5.3: Violin plots of delay in adding publications. Delay is given in full years. The width of each figure represents a kernel density estimation of the data. The bar inside the figure shows inner and outer quantiles. The red dot marks the mean delay.

the majority of conferences has been stagnant since 2002. Stagnant journals and venues are either discontinued or DBLP failed to notice new publications. The burst of new journals in 2011 coincides with a reduction of discontinued journals. I.e., after some delay, papers were added again to those journals. Again, this coincides with the increased funding of DBLP.

The growth of DBLP is not limited to adding recent publications. Figure 5.3 shows how the delay of adding publications developed over time. If a publication is added in the year of publication, the delay is 0. A delay of 10 means the paper is added 10 years after publication. Journal publications are sometimes added a year before their official publication date because of a long journal backlog. These publications have a delay of -1. Figure 5.3 shows the density of the distribution together with inner and outer quantiles and the median. Between 1995 and 2004, a significant number of publication was delayed. Between 2005 and 2011 most new publications were from the same year. In recent years, the number of delayed publications has increased again.

5.1.2 Evolving Topic Coverage

The previous section showed that the number of publications, venues and authors in DBLP has increased constantly over the years. However, these figures do not explain the nature of the growth. For example, in 2008 Laender et al. [LdLM+08] noted that DBLP does not cover all sub fields of Computer Science to the same extend. This bias is relevant for studies that derive general results for computer science based on the data in DBLP. The coverage bias also affects the use of DBLP for other applications. It is reasonable to assume that the author profiles provided

by the project are in some way used to measure the work of scientists. Authors form poorly covered fields will be underrepresented and important conferences or central papers might not be found in the collection. In this section, we study how the coverage of different topics in DBLP developed over time. The findings we discuss here were originally presented at ECDL 2010 [RH10a] and is joint work with Oliver Hoffmann. The results were obtained for an observation framework starting in October 1995 and ending in September 2009.

Table 5.1: The 27 topics based on Laender et al. [LdLM+08] and Martins et al. [MGLP09].

	Group description
t_1	Algorithms and Theory
t_2	DB, Information Retrieval, Digital Lib. and Data Mining
t_3	Computational Biology
t_4	Applied Computing
t_5	Comp. Graphics, Image Processing and Computer Vision
t_6	Integrated Circuits Design
t_7	Software Engineering and Formal Methods
t_8	Geoinformatics
t_9	Computer Education
t_{10}	Artificial Intelligence
t_{11}	Human Computer Interaction
t_{12}	Programming Languages
t_{13}	Multi-thematic
t_{14}	Operational Research and Combinatorics
t_{15}	Comp. Networks, Distributed Systems and P2P Systems
t_{16}	Simulation and Modeling
t_{18}	Web and Multimedia and Hypermedia Systems
t_{19}	Games and Virtual Reality
t_{21}	Information Systems
t_{22}	Machine Learning
t_{23}	Robotics and Control and Automation
t_{25}	Security
t_{26}	Comp. Architecture, High Performance Sys. and OS
t_{27}	Embedded, Real Time and Fault Tolerant Systems
t_{28}	Ubiquitous Computing
t_{29}	Formalism, Logics and Computational Semantics
t_{30}	Natural Language Processing

In a first step, we need to discuss what is a topic of computer science. In this work, a topic is a set of publications that are thematically related. By this definition, a paper can belong to one or more topics. There are several groups that attempt to determine the topic of papers automatically. Boyack et al. [BND+11] applied clustering algorithms to subject headings and words extracted from titles

and abstracts of 2.15 million MEDLINE publications. Fried et al. [FK13] present a similar approach for DBLP which also generated thematic *maps* of computer science. Smeaton et al. [SKG+03] also included relations of authors when computing the thematic development of the SIGIR conference. Using these approaches for our study is difficult as we do not have abstracts, keywords or subject information for the publications. We also need topics which are present through large parts of the observation framework. The approaches described above tend to create very specific clusters of publications. These clusters are often time specific as very specific topics emerge and decline within a few years. For our study, we need broad enough topics that exist for a long period of time so that we can observe their development. We also need a gold standard against which we can evaluate the coverage of DBLP. To determine how well a topic is covered we also need to know which parts of it are *not* listed in DBLP.

In this work, we will use a topic framework which was introduced by Laender et al. [LdLM+08] in 2008 and refined by Martins et al. [MGLP09] in 2009. Instead of categorizing individual papers, Laender et al. assign topics to conferences. All publications of this conference inherit this topic. We assume that a topic is well covered in DBLP if the collection lists most of the relevant conferences from this topic. The framework consists 27 topics lists of 1000 conferences that were assigned to these topics. Table 5.1 lists the names of the topics. We use the numbers provided in the original publications. Note that t_{17}, t_{20} and t_{24} are missing from the framework provided by Laender et al.. Unlike many lists of conferences which can be found on the Internet, the creation of this framework is well documented. The framework was created by analyzing the publications of Brazilian computer scientists between 1954 and 2007. It was refined and completed by polls among researchers. Conferences are also classified by scientific quality. International conferences were preferred over local meetings and those with low reputation ratings were excluded. The most significant drawback is that Laender and Martins do not consider journals or monographs.

Table 5.2 lists central figures for the conference framework. L_i is the set of conferences which are classified with topic i. Each conference has a single topic. However, 17 conferences are not assigned to a specific topic but classified as *multi thematic* (t_{13}). In a first step, we created a mapping between the list of Laender and Martins and the DBLP collection in September 2009. D_i denotes the set of conferences which we could map. The others were not present in DBLP at that time or we could not find a clear mapping. We can now simply define the coverage of a topic i as $cov = |D_i|/|L_i|$. There are significant differences in the coverage of topics. For example, while almost all conferences from t_2 are contained in DBLP two thirds of t_9 are missing.

This approach ignores the number of publications of an individual conference. This is a potential threat to validity as the number of publications per conference varies greatly. E.g., there might be a large conference which represents almost all papers of a topic as well as some small conferences. If DBLP lists the small conferences but not the big one the topic has a high *cov* value though most publications are missing. However, it seems unlikely that DBLP would add small conferences but systemat-

Table 5.2: Relevant values for the framework of Laender et al.[LdLM+08] and Martins et al. [MGLP09]. See Table 5.1 for a description of the topics.

	$\|L_i\|$	$\|D_i\|$	**cov**	**miss.**	**duplicate**	$\|L'_i\|$	L_i^{1995}	L_i^{2005}	cov_{2009}
t_1	42	33	78.6%	1 (2.4%)	1 (2.4%)	41	26	41	0.79
t_2	45	43	95.6%	1 (2.2%)	2 (4.4%)	43	24	43	0.96
t_3	12	10	83.3%	0	0	12	3	11	0.83
t_4	33	16	48.5%	2 (6.1%)	1 (3.0%)	32	15	32	0.48
t_5	69	52	75.4%	1 (1.4%)	0	69	36	66	0.75
t_6	46	33	71.7%	0	0	46	34	46	0.72
t_7	73	60	82.2%	1 (1.4%)	2 (2.4%)	71	31	70	0.82
t_8	13	8	61.5%	1 (7.7%)	1 (7.7%)	12	6	12	0.62
t_9	24	9	37.5%	2 (8.3%)	0	24	12	24	0.38
t_{10}	49	39	79.6%	0	0	49	22	47	0.80
t_{11}	27	26	96.3%	0	1 (3.7%)	26	20	26	0.96
t_{12}	41	36	87.8%	0	0	41	23	41	0.88
t_{13}	17	14	82.4%	0	2 (11.8%)	15	7	15	0.82
t_{14}	28	9	32.1%	1 (3.6%)	2 (7.1%)	26	12	25	0.32
t_{15}	82	67	81.7%	1 (1.2%)	3 (3.7%)	79	36	78	0.82
t_{16}	16	12	75.0%	0	0	16	10	16	0.75
t_{18}	43	39	90.7%	1 (2.3%)	2 (4.7%)	41	12	41	0.91
t_{19}	27	16	59.3%	0	0	27	4	24	0.59
t_{21}	14	11	78.6%	0	1 (7.1%)	13	7	13	0.75
t_{22}	38	33	86.8%	1 (2.6%)	0	38	21	38	0.87
t_{23}	40	16	40.0%	1 (2.5%)	1 (2.5%)	39	24	37	0.40
t_{25}	39	32	82.1%	0	0	39	15	38	0.82
t_{26}	60	53	88.3%	2 (3.3%)	3 (5.0%)	57	38	57	0.88
t_{27}	25	22	88.0%	0	1 (4.0%)	24	11	24	0.88
t_{28}	31	28	90.3%	3 (9.7%)	1 (3.2%)	30	2	30	0.90
t_{29}	34	31	91.2%	0	0	34	23	34	0.91
t_{30}	32	18	56.3%	2 (6.3%)	0	32	15	30	0.56
all	1000	766	76.6%	21 (0.2%)	24 (0.2%)	976	489	949	0.77

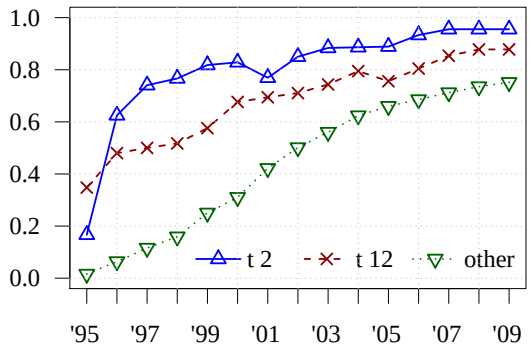
Source: Adapted from Reitz et al. [RH10a]

ically ignore large ones in the same topic: (1) For most topics L_i is large enough so that single conferences will not make up a very large part of the publications. (2) Large conferences are often more visible than small ones. We can assume that they are added first. In particular, as Laender et al. already excluded low quality and regional venues which are often missing from DBLP. It is also often difficult to determine the precise size of a conference, especially for those not listed in DBLP. E.g., many conferences have satellite events (collocated conferences, workshops ...). In many cases, publications from these satellites are not listed in DBLP and it is unclear if Laender et al. intended to include them in their framework. We also assume that a conference is covered if at least one publication is listed in DBLP. This is a small restriction as (1) DBLP lists all publication of a proceedings (within limits, see satellite events) and (2) DBLP continues adding new proceedings of a conference once it has been added.

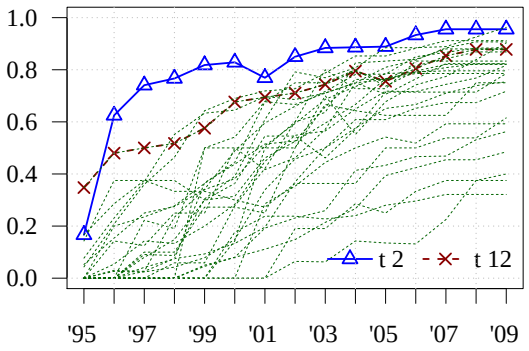
To compute the coverage for past times, we need to know when a conference was established and when it was added to DBLP. The date of the first venue is important because the conference cannot be listed in DBLP before that. We extract this information from digital libraries and conference websites. If conferences split, merge or change their name, it becomes difficult to tell when they started. If in doubt, we listed the first probable year as start year. Column *miss.* in Table 5.2 lists the number of conferences for which we could not find a start date. For these conferences we assume a start date of 1995, the beginning of our observations. This way we underestimate the coverage of DBLP. We have to consider that DBLP and Laender et al. use a different granularity to determine what an independent conference is. For example, the *International Middleware Conference* is listed by Laender et al. as well as the associated workshops *MPAC* and *MGC*. However, all three entities share the DBLP key `conf/middleware`. With only one key, it is difficult to map list entries and papers and tell when they were added. We count these conferences only once on L_i and D_i. The column *duplicates* in Table 5.2 shows the number of keys we ignored per topic. Column $|L'_i|$ shows the number of keys after duplicates have been removed. We can now define the coverage at the end of a specific year:

Definition 5.1 (Topic Coverage): Let L_i^y be the set of conferences from t_i that were established in year y or before. Similarly, let D_i^y be the set of conferences from t_i which were listed in DBLP at the end of year y. We can define the **(topic) coverage** at the end of a specific year as:

$$cov_y(t_i) := \frac{|D_i^y|}{|L_i^y|}$$



(a) t_2 and t_{12} compared to the combined coverage of all other topics



(b) t_2 and t_{12} compared to the individual coverage of all other topics

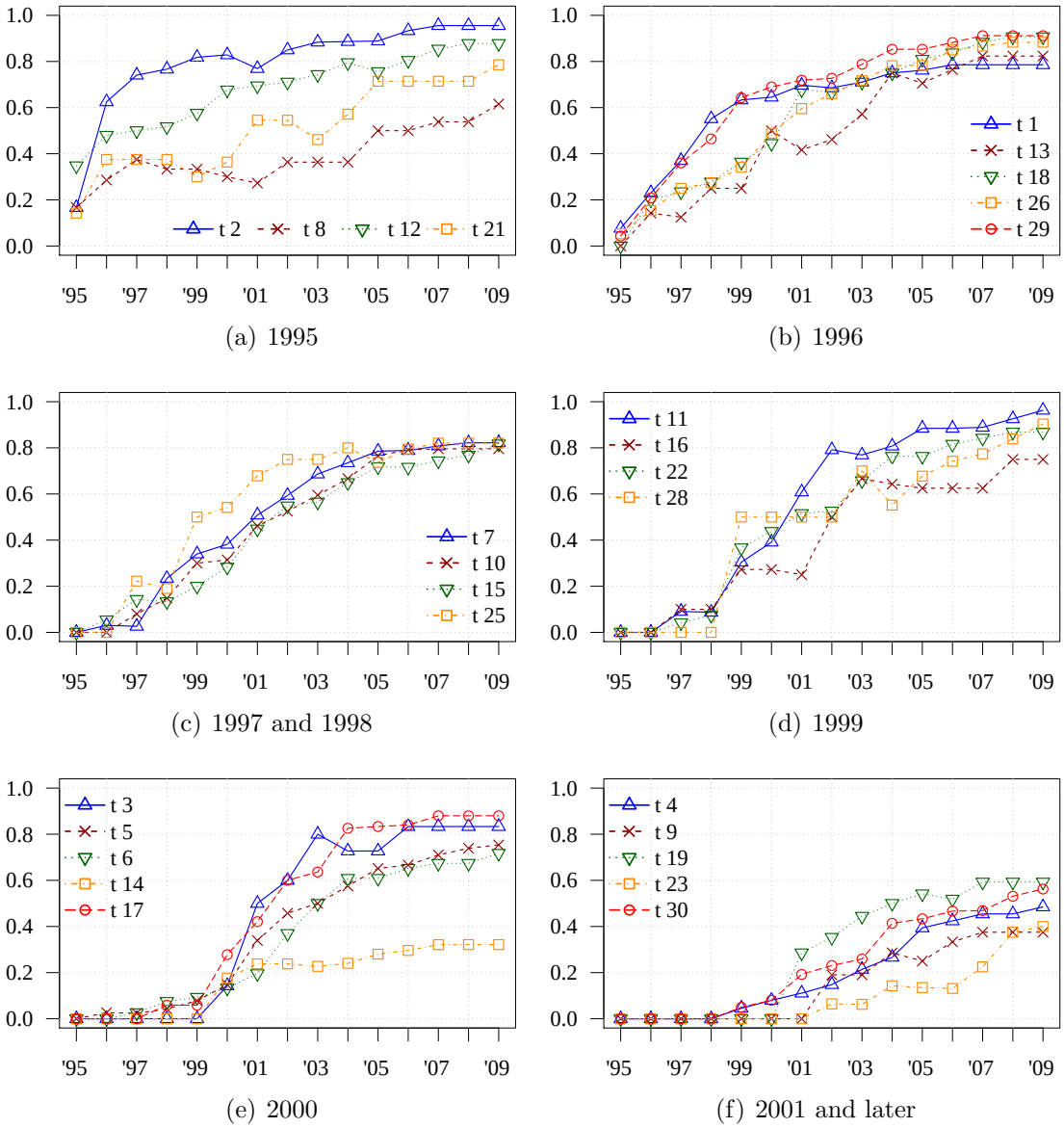
Figure 5.4: Coverage of t_2 and t_{12} compared to the coverage of the other topics. (X-axis: year and the Y-axis: coverage).

Source: Adapted from Reitz et al. [RH10a]

Note that cov_y does not monotonically increase over time but can decrease when uncovered conferences become relevant for L_i^y. We compute cov_y for all years between 1995 and 2009. For the year 2009, we must consider that the conference framework was already published and might have influenced the growth of DBLP.

Many topics had a low coverage in the 1990s which has improved over time. For example, consider the *International Conference on Very Large Data Bases* (VLDB, t_2) and the *IEEE CS Conference on Computer Vision and Pattern Recognition* (CVPR, t_5). Both conferences are central to their respective topics and were established before the DBLP project started. However, VLDB has been listed in the collection since the very beginning while CVPR was not added before April 2003. This is not a coincidence. When DBLP was founded, the focus of the project was on Databases (t_2) and Logic programming (part of t_{12}: Programming Languages) [LR06]. In fact DBLP was an acronym for *DataBases and Logic Programming*. Figure 5.4(a) shows the coverage of topics t_2 and t_{12} in comparison to the average coverage over all topics. Figure 5.4(b) compares the coverage of t_2 and t_{12} with all topics individually. We can see that (1) the coverage of t_2 is above the coverage of other topics, especially before the year 2002 (2) the coverage of t_{12} is low compared to t_2. This was to be expected because logic programming is only a small part of t_{12}. (3) While some topics reached a coverage similar to t_2, several topics have a coverage below 0.4. Column cov_{2009} in Table 5.2 lists the coverage at the end of the observed period. The five topics with the lowest coverage are t_{14} (Operational Research and Combinatorics, 32%), t_9 (Computer Education, 38%), t_{23} (Robotics and Control and Automation, 40%), t_4 (Applied Computing, 48%) and t_{30} (Natural Language Processing, 56%). Many of these topics lie on the edge of computer science which might give them a low priority for extension.

Figure 5.5 gives a detailed view on how the coverage of each topic has evolved. To improve the readability, we split the 27 topics into six groups depending on the year



(a) 1995 (b) 1996

(c) 1997 and 1998 (d) 1999

(e) 2000 (f) 2001 and later

Figure 5.5: Coverage values of different topics. The division in different figures is based on the first year the topics reached a coverage of at least 0.1. (X-axis: year and the Y-axis: coverage).

Source: Reitz et al. [RH10a]

they first reached coverage of at least 0.1. Note that topics t_3, t_8, t_{13}, t_{16} and t_{21} have less than 20 conferences assigned to them. I.e., small changes to L_i^y or D_i^y have a strong influence on the coverage. If we consider the coverage in 1998, we see that the scope of DBLP was still focused on t_2 and t_{12}. Aside from these topics, only t_1 had more than half of its conferences listed. Nine topics did not have any entry at all. The mean coverage at that point was 0.183 with a high standard deviation of 0.205, which also underlines the large differences. Contemporaneous with the increase of new records and conferences we discussed in Section 5.1.1, the coverage of several topics started to rise after the year 2000. The number of new papers for conferences already listed remained more or less constant and most of the additional new records could be used to widen the scope. The increase of coverage for most of the topics in Figure 5.5(c) and Figure 5.5(d) was rapid compared to the improvement of topics listed in Figure 5.5(a) and Figure 5.5(b). All topics which reached a coverage of 0.1 after 2000 did not reach a coverage of 0.6 or more at the end of the observation period. As stated above, these topics are not part of core computer science and it is therefore unlikely that they would be fully covered.

5.1.3 Relation Between Topic Communities

Conferences are not isolated from each other. One relation between conferences a and b is their common community, i.e., the set of all authors who published on a and b. When we consider this relation for a newly added conference, we find that usually more than 30% of the authors of new records have been listed in DBLP before. We assume that this integration into the existing collection is one criterion for a conference to be added to DBLP. If a topic has a low coverage, a new conference must integrate with conferences from other topics. We assume that an increasing coverage requires a good integration with other topics, at least at the beginning. To analyze this, we computed the shared coauthor relation between all topics.

Let A_i be the set of authors who published papers at a conference which belongs to topic t_i. We can define a graph where the topics are nodes and the edges represent the shared coauthor relation. Two topics are in relation if there is at least one common coauthor. The weight of the edge between topics t_i and t_k is defined as the reciprocal of the Jaccard index

$$J(t_i, t_k) = \frac{|A_i \cap A_k|}{|A_i \cup A_k|}$$

We compute the coauthor-relation graphs $g^{1996}, \ldots, g^{2009}$ for each year between 1996 and 2009. We do not consider the graph for 1995 as it contains only very few nodes and edges. We omit nodes for topics with a coverage of 0. The first graph which contains all nodes is g^{2002}. Beginning with g^{2004} the networks are complete but the edge weights are unequally distributed.

To analyze the structure of the individual graphs and the role of each topic, we compute how central each node is. The *betweenness centrality* [Fre79] $C_B(v)$ is based on the assumption that only the shortest paths between two nodes are relevant. If a node v is on a large number of shortest paths between all pairs of nodes, it is central for the graph. We compute

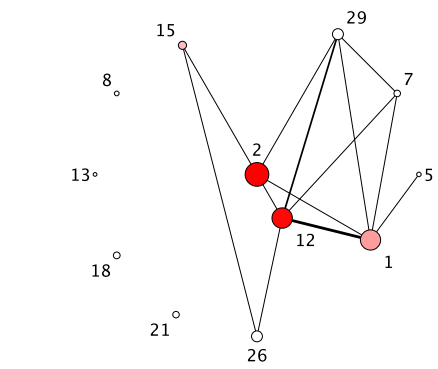
$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad \forall s, t \in V$$

where σ_{st} denotes the number of different shortest paths between nodes s and t and $\sigma_{st}(v)$ is the number of those paths which pass node v. We only consider edges with a weight > 0.1. Figure 5.6 shows graphs from six different years. To draw these graphs, we use a centrality layout based on the betweenness centrality which means that central nodes are positioned close to the center of the drawing. The lightness of the node coloring codes the C_B value. Dark colored nodes have a high centrality while white nodes have a betweenness centrality of 0. Nodes with a strong common community are drawn close to each other. The size of the node denotes the size of the respective community – i.e., the number of authors – and the thickness of the edges the strength of relation. To improve the readability, we do not draw edges with a weight less than $thres_e$ of the maximum where $thres_e$ varies between 10% and 30%. All size information is relative to the respective graph.

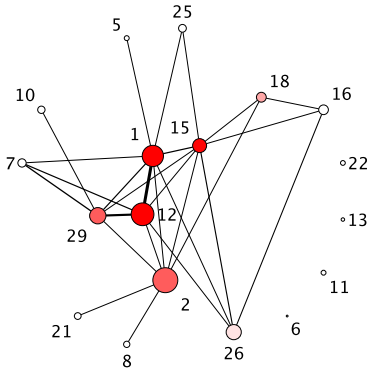
We saw in Section 5.1.2 that there are a number of topics which were established early besides t_2 and t_{12}. However, except for t_1 and t_{15}, none of them is central for g^{1996} in the sense of C_B. t_2 and t_{12} have similar centrality and community size but only a comparably small intersection. Both topics are connected to all other topics with a coverage > 0. While t_{12} achieves its centrality by a strong connection to a small number of other topics, t_2 has an even connection to the other nodes. This property can be found for all graphs.

In the following years, the central component of the graphs is formed by topics t_1, t_2, t_{10}, t_{12}, t_{15}, t_{26}, t_{29} and after 2002 t_{18}. This indicates that the communities of these topics are now represented in DBLP and are not just an extension of the communities of t_2 and t_{12}. The graphs have a clear outer zone in respect to centrality. In 2009, 13 topics had a betweenness centrality of 0. Among those topics are the topics with low coverage. The set of least integrated topics is stable. Between 2004 (the first graph where all nodes were present) and 2009, t_3, t_8, t_9 and t_{30} were always the least integrated topics with respect to C_B and d.

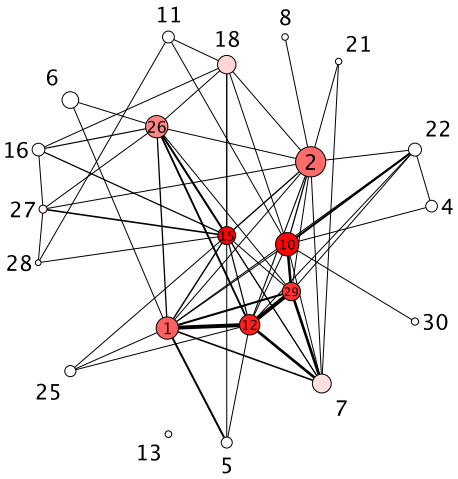
We must consider here that the communities of these topics are small and therefore intersections with other communities tend to be small as well. However, this also supports the assumption that a topic community needs to be partially present in DBLP to ensure that the coverage increases. To some degree, the graphs in Figure 5.6 show the increasing number of interdisciplinary collaborations. However, the observed time frame is only 15 years long and we assume that changes to the scientific community during this interval are not as large as observed here.



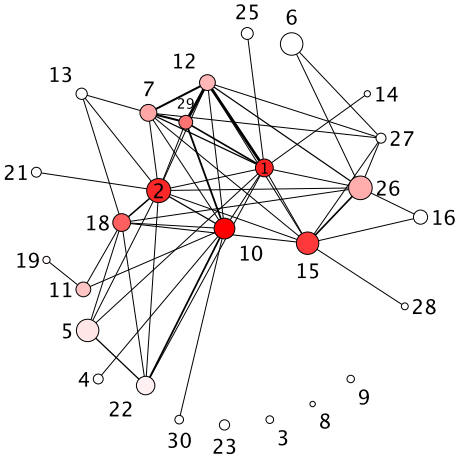
(a) 1996 $thres_e = 10\%$



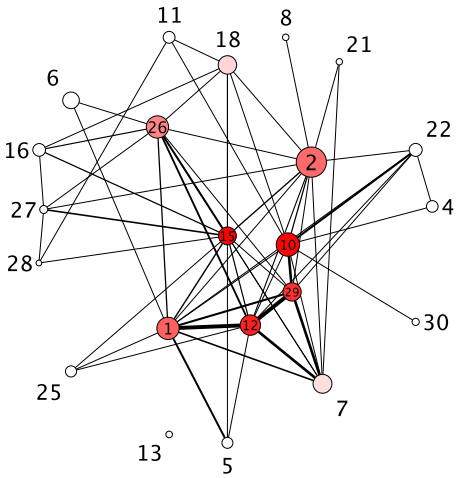
(b) 1997 $thres_e = 10\%$



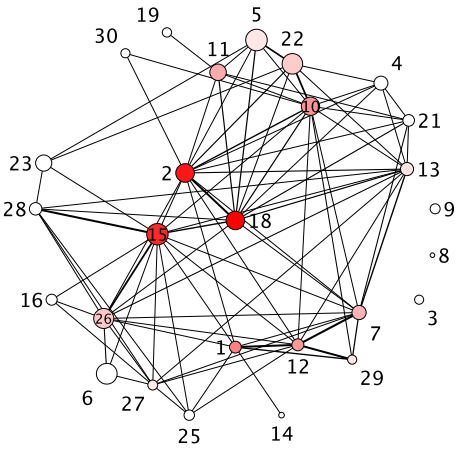
(c) 1999 $thres_e = 10\%$



(d) 2002 $thres_e = 20\%$



(e) 2005 $thres_e = 20\%$



(f) 2009 $thres_e = 30\%$

Figure 5.6: Relations between topics at the end of selected years.

Source: Reitz et al. [RH10a]

5.2 Understanding Error Reports

In Chapter 4 we discussed the importance of metadata quality for digital library projects. We saw that there are many different ways to detect an error, for example using algorithms or manual inspection. In this section we study the contribution of users to error detection in DBLP.

There are several open digital libraries which allow their patrons to edit metadata records or which provide interfaces which allow error reports from inside the application [Bov06]. For example, the United States Library of Congress published a set of images on Flickr with the request for corrections and additional information. Response to this call were considered to help improve the data quality [ZA10]. zbMath – a database for publications in Mathematics – provides an interface that helps users to report defects [MMR14]. Users can recognize defects that are difficult to find automatically. Consider a synonym defect where a single author is represented by multiple surface forms in the collection. We saw in Section 4.5 that many of these cases provide very little information that an algorithm can use. However, the author who is affected can spot the defect easily as well as colleagues and coauthors. Users also profit from correcting defects. Removing defects makes the collection easier to use for everyone. However, there is also a possible personal gain for authors to report defects related to their publications. E.g., McKay et al. [MSP10] found that a clean author profile is considered important by many researchers.

DBLP provides no specific interface to report defects. The only way to report a defect is by sending an email. In this section, we describe a study based on the emails sent to DBLP between January 2007 and November 2010. From the mails, we automatically extract error reports. We compare error reports to corrections made to DBLP and describe the impact of these corrections to DBLP. The study was published at TPDL 2011 [RH11]. It is joint work with Oliver Hoffmann.

5.2.1 Emails

The mail corpus contains messages to the DBLP project received between January 2007 and November 2010. Most spam and personal messages were removed, but the corpus still contains mails which are no error reports. The majority of mails are written in English or German. We excluded 23 mails which we could not open. We ignored any message part which is not plain text or HTML. This includes attachments. To make sure that we did not miss important information, we manually checked 100 random attachments but found no error reports. Overall, we retain 6311 mails, 1580 received in 2007, 1705 in 2008, 1664 in 2009 and 1362 in 2010.

To learn more about the mail corpus, we conducted a manual examination of 1000 randomly selected messages. In this subset, we found 458 messages which were intended as error reports. At this stage, it is not relevant whether a mail actually caused modifications to DBLP. We also consider mails which request the update of author information as error report. DBLP author profiles can contain affiliations

and web links for an author. These information can become outdated. Name-related inconsistencies are the most important type of reported errors. There are 217 mails reporting a *synonym*. A *homonym* is reported 92 times. 28 mails report more than one defect. One mail reported nine distinct defects at once.

DBLP does not specify a format in which error reports may be submitted. Not surprisingly, there is a wide variety of styles in the mail corpus. Below, we see four typical examples generated from real emails. They all differ in the quantity of given personal name information (underlined) and record metadata (printed in bold).

m_1	Please have a look at authors John Doe and John A. Doe they seem to be the same person and should be merged. Thanks J.
m_2	There is a paper **My title** published in **2005** at **venue** by **J. Doe**. The year is wrong, it should be 2006.
m_3	Please have a look at paper **conf/venue/XYZ09**. I think the author names are wrong.
m_4	There is an error in my 2nd publication. Please have a look at it, John

We considered the way in which persons (or more precisely: author profiles) and records are referred to. This is important as the user is not guided by policies or assisted by a user interface. There are different ways to refer to a person. Each person in DBLP has a unique key which can be found in the URL of the respective publication list[1]. For *Florian Reitz* the key is *Reitz:Florian*. The name itself can be given as full (*Florian Reitz*) or in abbreviated form (*F. Reitz*). In the order *key – full – abbreviated*, the names become more ambiguous. If a report is about an author profile, 50% of them are identified by the person key and 74% by the full name. Note, that DBLP stores abbreviated names for many publications so using them is not necessarily a result of a careless submitter. Using the key for other names like coauthors and other profiles is less common. Person keys are usually given as part of the full URL of the author profile.

References to records are less frequent than references to persons and can be found in 44% of the reports. This is not surprising given the large number of name-related defects which are not always related to a single publication. As for persons, DBLP defines keys for records. If a report contains a reference to a record, the most common type of information is the combination of *title + authors* (61.7% of references to records). Record keys are less frequently used than person keys. They can be found in 6.7% of all mails. Person keys appear in the URL of author pages while viewing record keys requires to follow a link. This might make them less perceivable for the user. Kapoor et al. [KBM+07] noted a similar reluctance to use unique identifiers – DOIs and ISBNs in their case – when storing records. However, in most cases, title and authors are sufficient to uniquely identify a record. Most submitters preferred to cite publications by using a text fragment containing all

[1]DBLP now also features a second key that is not generated from the name (e.g. 35/7092 for Author Wei Wang). This key was not readily available at the time of the study.

Table 5.3: Automatically identified person and record entities with different filters.

(a) person

	category	**detected**	**mails**	
P_1	*full*	14252	4618	(73.2%)
P_2	*partial*	8389	2853	(45.2%)
P_3	*abbreviated*	37051	3430	(54.3%)
P_4	*key*	3101	1783	(28.3%)
P	union	59656	5619	(89.0%)

(b) record

	category	**detected**	**mails**	
C_1	*publication key*	1215	423	(6.7%)
C_2	*title + full name*	5305	1155	(18.3%)
C_3	*title + partial name*	138	106	(1.7%)
C_4	*title + abbrev. name*	98	45	(0.7%)
C_5	*title + name key*	1464	561	(8.9%)
C_6	*title + year + pages*	4453	980	(15.5%)
C	union	6973	1636	(25.9%)

Source: Reitz et al. [RH11]

relevant metadata. Machine readable metadata were seldom used, mostly in form of HTML fragments and BibTeX. Most HTML fragments were copied from the DBLP web page. No report used the XML representation for records provided by DBLP.

A more significant problem is the use of relative information. In mail m_4, for example, it is not clear what exactly the *2nd publication* is. We found references like this in 60 of the 1000 manually checked mails. Most common are the DBLP publication numbers which we detected in 40 mails. These numbers enumerate the publications on an author page. These numbers are not stable. As a result of this evaluation, we started to plan a more restrictive error submission web site. On the one hand, it will force the submitters to give more detailed information but on the other hand, it will also assist them by proposing complete metadata or personal names.

To get information on the whole mail corpus and prepare further steps of our analysis, we automatically examined persons and records in all mails. We used the historical DBLP data set to generate two authority lists which contain the names of all persons listed in DBLP and all records at a specific time respectively. With these lists, we searched for person entities and records in the corpus. Table 5.3 lists the number of detected entities for the specified settings. We ignored names which are part of other names like *Michael Le* $\subset$ *Michael Ley* as well as very frequent names. In addition to abbreviated names, we also consider *partial* names which are

Table 5.4: Overview of edits and changes between January 2007 and December 2010. Column *type* shows the number of data elements per type in DBLP at December 2010. Column *wait* shows the expected time between two edits of the same data element based on *occurrence*.

type	occurrence		edits		changes		wait
author	3,150,610	(34.3%)	99,865	(53.6%)	70,546	(57.4%)	45,073
ee	795,374	(8.7%)	38,872	(20.8%)	38,132	(31.0%)	29,242
note	13,027	(0.1%)	12,570	(6.7%)	1,149	(0.9%)	1,480
title	1,432,869	(15.6%)	10,138	(5.4%)	2,226	(1.8%)	201,813
url	1,148,850	(12.5%)	7,550	(4.0%)	7,544	(6.1%)	216,764
others	2,372,346	(28.8%)	17,469	(9.4%)	3,368	(2.7%)	239,995

Source: Reitz et al. [RH11]

like full names without middle name parts and name extensions like *sen.*. The first column of each table gives the name of the setting. Abbreviated names are highly ambiguous. We will not use them in our study because they produce a high number of erroneously identified person entities.

5.2.2 The Reports

To get a global view on the user contributions, we must determine whether a mail is an error report or not. The manual evaluation covers only a fraction of the mail corpus. To get information for all messages, we apply an automatic identification algorithm based on the person entities and records we automatically detected. We then analyze the reports and discuss their significance to the quality management of DBLP and the properties of the repaired defects.

Automatic Detection of Error Reports

To find reports, we apply a simple heuristic. Consider mail m_1 from Section 5.2.1, which reports a name-related inconsistency. Fixing this inconsistency requires changes to author elements which are related to person entities *John Doe* or *John A. Doe*. If we can find such corrections in the set of all modifications, we call them *triggered by* m_1. We examine modifications which can be detected up to 30 days after the mail was received. For the purpose of this study, we use the date the mail was received by DBLP instead of the send date. This should remove effects of wrong send dates (e.g. wrong time zone setting on sending server) but introduces a minor delay. However, in most cases, the delay between sending and receiving should be smaller than one minute. Column *wait* in Table 5.4 shows the expected number of days between two edits to the same data element. For all types of data element this

number is significantly higher than 30 which means that an unrelated edit during the first 30 days after receiving the mail is unlikely. We call a change *triggered* by a mail if at least one of its edits is triggered. For detected triggered modifications to records, we examine all changes to the data elements of the respective metadata record. For a person p, we consider all records of publications authored by p, but we register only changes which have the name of p as *old* or *new* value. There are two reasons for doing so: (1) Some persons have many publications so we examine changes from a large number of data elements which increases the chance of random hits. (2) If only a personal name is given, it is likely that the defect is related to the person and not to a local property of a specific publication, e.g., a typo in the title. We also examine any modification to the author's personal record.

We detected 7996 triggered edits, which is 4.3% of all edits in the period January 2007 to December 2010. 7486 edits were detected by considering person entities and 1536 by considering record entities. The number of triggered changes is 6195, which is 4.91% of all changes in the observed period. The average waiting time between receiving the report and the edit is 1.28 days if an edit is triggered by record data and 1.52 days if it is triggered by a person reference. In any case, most reports were processed during the first 48 hours after being received.

Of all mails, 47.2% triggered edits or changes, which is slightly more than we obtained from the manual analysis. For 77 (16.9%) mails, which we manually identified as reports, we could not find triggered edits. Of these mails, 17 mails were meant to be reports but the requested modifications were defective. 16 mails provided insufficient information for the automatic detector. For the remaining 44 mails, we could not find related edits. It is unclear why these mails were not processed. 68 mails manually labeled as non-report caused modifications (12.5%). 80% of the non-reports triggered changes to the author record, i.e., they added or modified web link or affiliation information. These data came from the mail signatures and were obviously updated as a side effect. Five mails were replies to earlier reports and 8 mails contained large lists of record data. It is likely that a change can be found for a large list.

Are Reports Significant for DBLP?

The absolute number of triggered edits and changes seems small compared to the total number of these modifications. However, the fraction of triggered edits and changes is significant for some data element types. Figure 5.7 shows the combination of edits by data element type for the different sets of detected entities. *All* is the union of sets P and C. The *author* element is particularly frequent for all filters. Of all edits in the observed time frame, 53.6% affected an *author* element. For the different filters, the fraction of triggered edits that affect an author element can be up to 97% (C_4). C_4 is the most ambiguous category of records. These records are listed with abbreviated names. For abbreviated names, name-related inconsistencies are much more likely than for records with full names. As a result, name defects are more frequent and their corrections are usually more urgent so there is a high number

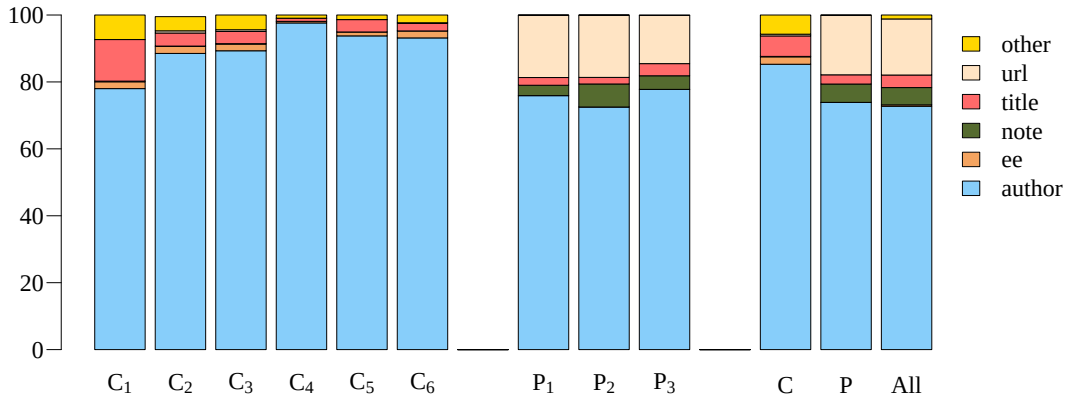


Figure 5.7: Distribution of edits by data element type for different filters (for filter names see Table 5.3). *author* also includes a small number of modifications to an editor element.

Source: Adapted from Reitz et al. [RH11]

of reports on this element type. We already pointed out that author elements are very relevant because the information of this data element determines the author profile on which a publication is listed. Most other data elements in DBLP are local, i.e., an edit to these element will only cause a slight modification to the appearance of the record. Modifications to *ee* – the second most frequent element type – are almost never triggered by mails. The *ee* element stores a link to the electronic edition of the publication, usually on the publisher page. *ee* elements are often modified in bulk without specific user input. For the observed time frame, we found that 95% of all edits to the *ee* element were done on just 65 days.

Table 5.5 lists the fraction of reported changes to the total number of changes for the most common data element types. The table also shows the fractions for each year in the study. The *url* element in particular has a high contribution. *url* is used in two different places in DBLP. However, modifications mostly occur to the *url* elements that contain the website displayed on an author profile. This information is not updated semi-automatically like the *ee* element. The *note* element is similar to *url*. It mostly contains the affiliation information which is displayed in the author profile.

Many triggered changes are related to the *author* or *editor* element. We can classify these changes according to our framework (see Section 4.2.1) into merge, split, distribute and rename changes. Using a preliminary version of the framework presented in Section 4.2.1, we detected 28,151 merges, 2,658 splits, 10,920 distributes and 13,228 renames for the observed period. Among the triggered corrections are 1,693 merges (6.0%), 417 splits (15.7%), 443 distributes (4.1% of all distributes) and 226 renames (1.7%). Usually, a synonym or homonym problem is related to a name but not to a single publication. Therefore, it is not surprising that corrections related to name-inconsistencies are far more often induced by personal names than by other

Table 5.5: Fraction of triggered changes by type for different years.

	author	**editor**	**ee**	**note**	**title**	**url**	**others**	
2007	776	9	6	32	20	270	16	1129
	5.1%	8.1%	0.2%	24.4%	13.3%	13.2%	6.2%	5.2%
2008	1225	20	17	148	27	424	27	1888
	8.0%	19.6%	0.7%	9.6%	3.2%	27.5%	3.7%	8.3%
2009	1098	14	6	127	62	401	26	1734
	5.6%	22.6%	<0.1%	5.5%	8.4%	17.4%	3.0%	3.7%
2010	969	8	5	103	489	1633	11	1444
	4.8%	4.7%	0.1%	26.3%	21.5%	14.9%	1.0%	4.2%
	4068	51	34	410	214	1338	80	6195
	5.8%	11.4%	0.1%	9.4%	9.6%	17.7%	2.7%	4.9%

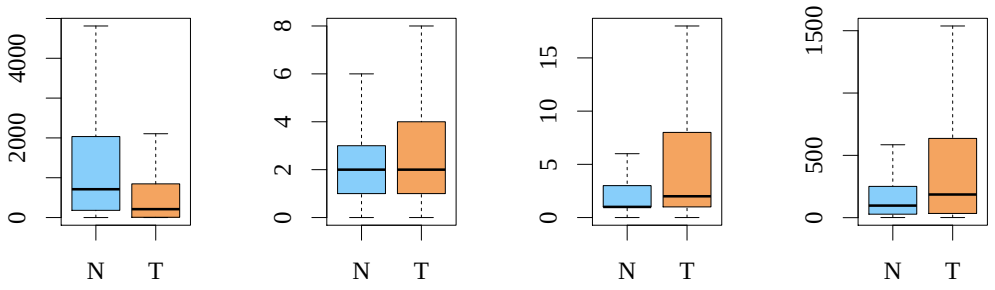
Source: Reitz et al. [RH11]

record metadata. The low number of renames was surprising as they cannot be discovered automatically. We considered renames more closely and found many cases where abbreviated names were changed to full names. This increase of information is usually provided by publishers and not by users.

Another class of defect is missing or surplus data. This type of defect is difficult to detect automatically because similarity or dissimilarity metrics do not work here. Again, reports related to the author element make up a significant fraction of add and remove edits. Of all edits that add an author element to a publication, 17.87% were triggered by an email, whereas the fraction was only 3.1% for those edits that remove an author element. We considered these cases more closely and found many cases where duplicate author elements were removed. This type of correction can occur automatically and does not need triggering from a user.

Do Triggered Edits Differ from Others?

In a next step, we compared the properties of triggered edits with properties of non-triggered edits. We use edits instead of changes as they allow consideration of individual records. At first, we examine the affected records. In general, triggered edits affect younger records than general edits. Figure 5.8(a) shows a boxplot for the absolute age of records, i.e., the number of days between adding the record to DBLP and the modification. If the edit is triggered, the mean record age is only half as high as if the edit is not triggered. Only few non-triggered edits affect records which are younger than 100 days. The median for triggered edits however is only slightly higher (211). Apparently, new records are more interesting for the users. We assume that many persons wait for their publications to appear and check them immediately. In Section 5.2.3, we will see further evidence for this. We expected



(a) age (days) (b) authors per record (c) publications per author (d) page impression

Figure 5.8: Boxplots of different properties for entities affected by triggered edits (T) and non-triggered edits (N). Outliers are not displayed.

Source: Reitz et al. [RH11]

to find more triggered edits for records with many authors. In this case, a defect appears on many author pages and more people can notice it. Though there is a tendency in this direction, this effect is small (Figure 5.8(b)).

Figure 5.8(c) shows differences in the number of publications per affected author[2]. The median and mean values of publications are twice and 3.8 times as high respectively for triggered edits as median and mean for non-triggered. Persons with many publications are usually more central to the field and therefore defects are detected faster. We also assume that many persons with a small number of publications already have left academics or are from fields weakly related to computer science. In the later case, DBLP is probably not considered a central tool and there is no incentive to report defects. To measure the visibility of an author record, we use the number of page impressions as a proxy. A page impression is the rendering of an author profile. We consider impressions for the same page from the same IP within one hour as one impression. We excluded IPs with a high number of impressions. These IPs probably belong to web crawlers. Figure 5.8(d) shows the number of page impressions logged on the main DBLP server between October 2007 and October 2010. Again, persons affected by triggered edits are more popular than others. However, 6.9% of these persons have ten or less page impressions. Note that number of publications per profile and number of page impressions for that profile correlate.

5.2.3 The Submitters

In a final step, we examined the persons who submitted the 2976 automatically identified error reports. Identifying these persons is not easy. Email addresses are usually unique but a person can have more than one. We tried to parse signatures

[2] $min = 0$ because of persons which were added during an edit and had no publication before.

Table 5.6: Report count by submitter. Column fraction list the fraction of reports by class of submitter.

reports sent	persons	listed DBLP		fraction of reports
1	1890	1357	(71.8%)	64%
2	275	218	(79.3%)	18%
3-5	92	69	(75.0%)	10%
6-20	10	9	(90.0%)	3%
137	1	1	(100.0%)	5%

Source: Reitz et al. [RH11]

which can be found at the end of many mails but there are many different designs and we were not able to extract reliable data. For this study, we use the sender name – an optional field which is provided with the mail address. We found 2268 distinct submitters. 1654 of these submitters are listed in DBLP as authors or editors. Note that we could not identify all submitters so the number of those who are listed in DBLP is probably higher. However, there is also a noticeable number of non-scientific staff or young PhD students who hand in reports. There are two extremes of user contribution frequency: (1) all reports are sent by a small group of heavy submitters, (2) many submitters send a single message each. DBLP tends towards the second type. Table 5.6 lists how the submitters are distributed by number of reports. 83% of the submitters handed in a single report. There are only 11 heavy submitters who sent more than five reports. The most frequent submitter sent 137 reports (5% of all reports). All heavy submitters are listed in DBLP except for one. This person is a non-scientific staff member who apparently is responsible for keeping records of a work group clean. Together the heavy submitters sent 8% of the relevant messages. Having a large number of infrequent submitters is positive as the project does not want to depend on a single person. On the other hand, we noted that the quality of reports is higher for frequent submitters. All heavy submitters usually used record or person keys if possible and – with the exception of one person – avoided the problems we discussed in Section 5.2.1.

We found that many error reporters are interested in the correction of defects which are related to themselves. This is not surprising: DBLP has become an important instrument to assess the work of scientists so there is a pressure to assure completeness and correctness of one's own entry. Authors also have a more detailed knowledge of their work which makes it easier to spot defects. 1364 reports (45.8%) triggered an edit to the name of the submitter, 1502 reports (50.5%) altered a publication authored by the submitter, 480 times, we observed an edit to the name records of a coauthor of the submitter. Altogether 1836 (61.7%) reports triggered edits directly or almost directly related to the submitter. For the heavy submitters, the number of self-related messages is much smaller, namely 15.6%. This is not surprising as it is unlikely that there are that many defects related to a single person.

Chapter 6

Conclusion and Future Work

The metadata of digital libraries have been the subject of many studies, from defect correction to content analysis. However, to the author's best knowledge, the history of those data has been mostly ignored. The goal of this work was to prove that – despite obstacles and restrictions – it is actually possible to *learn from the past* of a digital library. I.e., we discussed how information of growth and change of metadata can be used to better understand a collection. In this chapter, we will summarize the central results of our work and discuss possible venues for future work.

The first step of this work was to confirm the availability of historical data from digital libraries. Though many projects do not retain or provide historical metadata, we were able to identify several collections with sufficient data. When we studied these data sets we found that many have properties that impede an analysis in one way or another. To mitigate these problems, we defined a dynamic metadata model (Chapter 2) which is not based on the time of a modification but on the observance of that modification. Based on this model we conducted a number of studies on the collections we identified. The central contributions of this work are summarized below:

Automatic extraction of modifications. (Chapter 3) Based on the dynamic metadata model, we defined a four step framework which extracts modifications from collections that provide historical metadata. We start with trivial local modifications and successively group them into larger modifications. The framework is designed so that data sets with different temporal properties can be handled. We implemented the first three steps of the framework and successfully applied it to collections of up to 48 million records.

Automatic detection and classification of defect corrections. Creation of test collections. (Chapter 4) We implemented a system to automatically extract certain types of defect corrections. To this goal, we identified patterns which indicate that a defect was removed. Most important, the system can identify three types of corrections to entity reference defects. We successfully applied the implementation

to the DBLP and IMDB collections. Based on the extracted defects, we defined two test collections. The case-based collection gives fine grained information on modifications but provides only limited context information for the defects. We used it primarily to study the properties of defects. The embedded test collection is more suitable for evaluation of entity disambiguation algorithms. Unlike classical test collections, the embedded collection is large enough to evaluate the runtime of algorithms. It provides full context information and a partial ground truth. The test collections harness the work which has been invested in the underlying collections during several years. Therefore, the tedious manual work which limits current test collections is not necessary. We made the test collections based on DBLP publicly available.

Properties of defects. (Chapter 4) We studied base properties of defects which are relevant for most state-of-the art person name disambiguation algorithms. At first we considered name pairs of synonym defects. We could show that similarity of the pairs varies between collections and that simple name based blocking strategies not always produce acceptable results. We also studied the neighborhood of defects in a local entity-relationship model. Our results show that for many defects, only a small amount of information is available. This is relevant as the amount of information influences the likelihood that an algorithm can detect a defect.

Exploratory studies on the use of historical data. (Chapter 5) We showed that historical data can be used to better understand other aspects of digital libraries. At first we presented a study on general growth and the thematic coverage of DBLP over time. We also showed how a specific type of user contribution – emails with error reports – influence quality insurance of DBLP.

Many aspects of this thesis are exploratory. In part, the studies we discussed are limited by the properties of the data sets we used. Another factor is that we focused mainly on DBLP. Future work will have to identify additional and better sources of historical data. We need to use these data to test how the modification and defect detection implementations can be adjusted to other collections. In particular, we need to determine how strongly our approaches are biased by the edit policy of the project behind the collection. One aspect of this problem that we need to address in the future is how to exchange historical metadata. None of the metadata formats commonly used for digital libraries can accommodate temporal data. Effort should be directed at

- Extending metadata exchange formats to accommodate historical data.
- Extending exchange protocols so that historical information can be shared with others.

As part of his diploma thesis, Wenlong Chen explored possible extensions of OAI-PMH [Che13] that would handle aspects of dynamic metadata. The protocol extension provides records as a sequence of revisions. Additional parameters were added

to the protocol to query based on internal time. Future work will have to determine if this extension is sufficient or if more detailed query parameters a warranted.

The test collections we provide contain a large amount of realistic defects for DBLP. However, it is unclear if the collection actually differs from known test collections for the ER problem. The primary challenge is the different evaluation scenario. Most current collections expect an algorithm to determine author profiles from scratch on a set of records. Our collection provides actual problems (such as merges) that the algorithms need to solve. We need to determine how existing ER algorithms can be modified to be tested fairly in this scenario. Once a number of algorithms has been adjusted, we need to evaluate them with our test collection. To this goal, we must also better understand the implications of a partial ground truth for an evaluation. It is currently unclear if the evaluation strategies we outlined in Section 4.3.5 are helpful.

We also need a better understanding on the properties of defects. The studies found in this work only touch on basic aspects such as availability of information and difference of the surface forms. Future work must consider more complex information such as motifs in the data graph or non-local graph properties that are typical for defects. In this work, we concentrated on DBLP which – from our point of view – is the most accessible of the collections we discussed. The study on surface forms of synonyms indicated that collections can differ significantly in properties that are heavily used by ER algorithm. Further studies need to consider these differences more closely. This is particularly important as ER algorithms are currently evaluated against a small set of test collections with the implicit assumption that results transfer to applications on different data sets.

Finally, we need to improve our understanding of some aspects of the modification detection framework. While we discussed redesign changes as the 4th step of our framework, we have not implemented an automatic extraction for it. In part, this is because of the simple and stable record structure of most collections we considered here.

Appendix A

Digital Collections used in this Work

Contents

This chapter provides technical details of the data sources for IMDB and Wikipedia. A general description and information on the other collections can be found in Section 2.4.

A.1 IMDB Input Data

The IMDB data set is provided as a set of files which contain information on a specific role or property in a movie or series. E.g., the file *actors.txt* contains the line

> ***Example A.1:*** Stewart, Patrick (I) 'Red Dwarf' (1998) (TV) [Himself] 18

where

- **Stewart, Patrick (I)** is the name of the actor (with homonym disambiguation number)
- **'Red Dwarf' (1998)** is the name of the TV-show with year for disambiguation
- **(TV)** is the type of production (here: TV-series)
- [**Himself**] is the name of the role
- **18** is the position in the credits

There are three types of input files:

- PER.: containing information on a role filled by a specific person (see example above).
- BOOL: true or false as properties of productions.
- TEXT: a text with some annotations.

We created parsers for all three formats. The data from these parsers is used to model the metadata records. We created records for each production identifier and each person identifier. Table A.1 shows the data elements we modeled for a production type record. *usage* denotes the percentage of records of a specific type.

Table A.1: Primary keys used for production type records. Lists only keys which are used in at least 100 records. † data element was not available during the whole observation period due to data defects of several files.

key	**usage**	**data**	**key**	**usage**	**data**
actors	73.4%	PER.	language†	44.9%	TEXT
actresses	64.1%	PER.	literature	2.5%	TEXT
aka-titles	8.7%	TEXT	locations	16.0%	TEXT
alternate-versions	0.5%	TEXT	miscellaneous-comp.	7.2%	TEXT
business†	7.9%	TEXT	miscellaneous†	26.9%	TEXT
certificates	9.2%	TEXT	movie-links	8.2%	TEXT
cinematographers	30.0%	PER.	plot	12.0%	TEXT
color-info	48.2%	TEXT	producers	50.0%	PER.
complete-cast	4.1%	BOOL	production-companies	36.3%	TEXT
complete-crew	2.0%	BOOL	production-designers	11.5%	TEXT
composers	25.4%	PER.	release-dates	84.6%	TEXT
costume-designers	10.3%	PER.	running-times	30.0%	TEXT
countries	48.6%	TEXT	sound-mix	18.6%	TEXT
directors	57.8%	PER.	soundtracks	3.7%	TEXT
distributors	27.6%	TEXT	special-effects-comp.	1.3%	TEXT
editors	31.5%	PER.	taglines	4.3%	TEXT
genres	35.4%	TEXT	technical	19.5%	TEXT
goofs	1.8%	TEXT	trivia	5.4%	TEXT
keywords	19.2%	TEXT	writers	49.3%	PER.

The following data element types are used in at least 100 different person type records. The percentage denotes how many records used this key at least once during the observed time.

- aka-names (55.9%) data type: TEXT
- biographies (63.1%) data type: TEXT

Note, that the person's name is used as identifier and not provided as a data element in the input files.

A.2 Wikipedia

This section lists the infoboxes extracted from the English, German and French Wikipedia versions. For each infobox the primary name is given. Some infoboxes have more than one name in the historic collection. ► denotes names which were used instead of the primary name in the past. ▷ denotes names which were used parallel to the primary name as a synonym. We only accepted synonyms that never had their own code and for which there is no independent user guide.

Key to the following tables:

- **pages:** Number of Wikipedia pages with this infobox.
- **rev.:** Average number of revisions per record.
- **keys:** Number of keys found for an infobox.
- **(*):** Number of keys which are used in at least 100 different records.
- **since:** First year the infobox was observed.

In the following tables, *IB* stands for *Infobox*

Table A.2: Most frequent infoboxes from English-Language Wikipedia.

		records			
No	**Basename**	**pages**	**rev.**	**keys (*)**	**since**
E01	Persondata	1,033,290	2.12	1963 (25)	2001
	▷ Personal Data				
E02	IB settlement	281,263	5.22	14941 (563)	2001
	▷ IB comarca				
	▷ IB county				
	▷ IB Department				
	▷ IB municipality				
	▷ IB regency				
	▷ IB district				
	▷ IB province				
	▷ IB state				
	► IB City				
E03	Taxobox	228,797	4.73	4891 (146)	2001
E04	IB album	125,119	16.29	6964 (98)	2001
	▷ Album infobox				

E05	IB person ▷ IB director	113,444	7.89	6898 (221)	2001
E06	IB football biography ▷ IB football player ► Football player IB	99,774	21.46	6457 (260)	2005
E07	IB film ► IB Movie	77,964	14.31	7875 (103)	2001
E08	IB musical artist ▷ IB band ▷ IB Musician ▷ IB singer	70,243	27.34	8807 (113)	2001
E09	IB single ▷ Single IB	49,801	18.58	2718 (89)	2005
E10	IB actor	43,847	15.24	3962 (116)	2001
E11	IB company	42,764	12.88	8840 (75)	2001
E12	IB NRHP	39,493	4.51	846 (74)	2001
E13	IB French commune ▷ Communefra ▷ French commune ▷ Communefra/Corse	36,847	7.53	794 (98)	2001
E14	IB book	29,134	8.16	1822 (35)	2001
E15	IB officeholder ▷ IB AM ▷ IB governor general ▷ IB MEP ▷ IB Minister ▷ IB MLA ▷ IB MSP ▷ IB premier ► IB President	28,468	12.76	4070 (106)	2001
E16	IB television	26,841	26.35	5793 (107)	2001
E17	IB ship characteristics	25,377	4.66	726 (64)	2004
E18	IB ship image	25,238	2.58	183 (3)	2004
E19	Geobox	24,077	3.69	2866 (466)	2001
E20	IB military person	23,473	10.99	1028 (50)	2001
E21	IB ship career	23,036	4.38	598 (51)	2004
E22	IB school	21,439	13.42	6736 (282)	2005
E23	IB UK place	19,840	5.57	1267 (83)	2001
E24	IB Indian jurisdiction ► IB Indian urban area	19,200	5.97	4880 (122)	2001
E25	IB radio station ► Radio station	18,625	12.06	1914 (40)	2001
E26	IB vg	18,578	17.38	2467 (72)	2001
E27	IB road	18,543	6.84	870 (98)	2005
E28	IB MLB player	17,973	16.06	1918 (54)	2001

E29	IB university	17,347	17.36	6537 (91)	2001
E30	IB football club ► Football club infobox	17,229	23.19	7415 (91)	2001
E31	IB television episode	16,147	8.41	582 (28)	2005
E32	IB writer ▷ IB author	15,690	11.86	1733 (62)	2001
E33	IB scientist	14,373	10.98	1348 (59)	2001
E34	IB military unit	14,072	9.67	1384 (78)	2001
E35	IB planet	13,237	4.11	589 (63)	2001
E36	IB mountain	13,187	7.63	883 (85)	2001
E37	IB river	13,172	2.36	564 (42)	2005
E38	IB German location ▷ IB Gemeinde in Deutschland ▷ IB Ort in Deutschland	12,950	4.85	526 (85)	2001
E39	IB airport	12,249	7.04	826 (90)	2001
E40	IB sportsperson	12,176	2.70	422 (63)	2008
E41	IB military conflict ► Warbox	12,127	31.54	1779 (51)	2001
E42	IB NFL player ► IB NFLactive	11,623	17.95	1202 (161)	2006
E43	IB ice hockey player	11,446	11.37	707 (44)	2001
E44	IB cricketer ► IB cricketer biography	11,078	7.01	1693 (213)	2001
E45	IB station ▷ IB NS-station	10,322	8.23	653 (147)	2007
E46	IB software	10,045	15.68	1600 (41)	2001
E47	IB body of water ► IB Lake	9,881	5.68	600 (48)	2001
E48	GNF Protein box	9,842	1.07	70 (40)	2007
E49	Geobox River	9,165	2.24	432 (101)	2007
E50	Chembox	8,551	11.16	1387 (192)	2001

Table A.3: Most frequent infoboxes from German-Language Wikipedia.

		records			
No	**Basename**	**pages**	**rev.**	**keys (*)**	**since**
D01	Personendaten	460,319	3.02	872 (9)	2003
D02	Taxobox	36,462	3.61	657 (59)	2005
D03	IB Film	22,257	8.13	1216 (49)	2004
D04	IB Fußballspieler ► Fußballspieler IB	20,988	12.94	398 (50)	2007
D05	IB Ortsteil einer Gemeinde in Deutschland ► IB Ortsteil einer Gemeinde	20,227	5.35	398 (50)	2007
D06	IB Unternehmen	15,021	7.80	1490 (32)	2005

D07	IB Gemeinde in Deutschland ► IB Ort in Deutschland	13,911	12.93	1006 (47)	2006
D08	IB Fluss	11,452	5.76	668 (118)	2006
D09	IB Gemeinde in Frankreich ► IB Ort in Frankreich ► IB französische Gemeinde	9,766	6.54	249 (42)	2005
D10	IB Band	9,225	8.59	1156 (96)	2005
D11	IB Gemeinde in Italien ► IB Ort in Italien	8,008	3.13	263 (36)	2006
D12	IB Berg	7,527	4.90	293 (40)	2004
D13	IB Eishockeyspieler	7,164	4.51	144 (57)	2007
D14	IB Ort in den Vereinigten Staaten	6,185	4.91	209 (41)	2007
D15	IB Musikalbum	5,992	6.52	454 (32)	2006
D16	IB Chemikalie	5,627	13.79	522 (56)	2007
D17	IB Burg	5,109	4.02	114 (23)	2007
D18	IB Fußballklub	4,817	12.37	717 (48)	2007
D19	IB Ort in Tschechien	4,524	4.14	121 (36)	2007
D20	IB Fernsehsendung	4,228	8.24	324 (28)	2007
D21	IB See	4,019	4.06	210 (43)	2006
D22	IB Asteroid ► Asteroid	3,970	4.30	125 (30)	2004
D23	IB französischer Kanton	3,925	3.82	20 (19)	2006
D24	IB Insel	3,609	3.42	97 (29)	2008
D25	IB Schiff	3,562	2.74	262 (95)	2006
D26	IB Ort in Polen	3,445	6.57	281 (62)	2006
D27	IB hochrangige Straße ► IB Autobahn	3,252	3.90	214 (41)	2009
D28	IB Ort in der Schweiz	3,195	13.12	225 (32)	2007
D29	IB Software	3,157	17.71	387 (27)	2005
D30	IB Römisch-katholisches Bistum	3,135	3.87	141 (53)	2007
D31	IB County (Vereinigte Staaten) ► IB County (USA)	3,089	7.22	82 (17)	2005
D32	IB Publikation	2,983	5.24	316 (30)	2006
D33	IB Schienenfahrzeug	2,876	5.86	413 (118)	2006
D34	IB Hochschule	2,617	6.19	408 (24)	2007
D35	IB Flugzeug	2,609	4.33	84 (9)	2006
D36	IB Stadion	2,576	2.86	98 (34)	2007
D37	IB Ort in Russland	2,416	3.94	144 (48)	2007
D38	IB Gemeinde in Österreich ► IB Ort in Österreich	2,403	11.90	484 (53)	2007
D39	IB PKW-Modell	2,383	9.15	167 (23)	2007
D40	IB Militärischer Konflikt	2,330	9.85	177 (28)	2006

D41	IB Ort in Portugal	2,322	2.10	71 (36)	2007
D42	IB Flughafen	2,286	7.94	398 (59)	2005
D43	IB Schule	2,073	10.20	494 (37)	2005
D44	IB Schiff/Antrieb	1,959	1.53	23 (7)	2010
D45	IB Gemeinde in Spanien	1,917	4.97	215 (40)	2008
D46	IB Computer- und Videospiel	1,894	19.43	330 (27)	2006
D47	IB Bahnhof	1,865	6.91	137 (44)	2007
D48	IB Sprache	1,832	6.76	176 (15)	2004
D49	IB Biathlet	1,812	4.04	59 (38)	2008
D50	IB Ort in der Türkei	1,717	4.56	127 (41)	2007

Table A.4: Most frequent infoboxes from French-Language Wikipedia.

		records			
No	**Basename**	**pages**	**rev.**	**keys (*)**	**since**
F01	Taxobox	77,927	1.68	112 (7)	2006
F02	IB Commune de France ► IB Communes de France	37,002	10.48	698 (55)	2007
F03	IB Musique (œuvre)	24,984	5.71	590 (88)	2008
F04	IB Footballeur	19,718	16.02	1403 (44)	2006
F05	IB Compétition sportive	18,954	5.43	529 (107)	2007
F06	IB Politicien	15,394	5.49	1458 (168)	2009
F07	IB Musique (artiste)	14,965	10.71	1071 (72)	2007
F08	IB Cinéma (film) ► IB Film	14,407	5.73	816 (58)	2005
F09	IB Cinéma (personnalité)	12,700	6.06	656 (40)	2008
F10	IB Biographie	12,289	6.19	1002 (47)	2006
F11	IB Société	10,977	10.70	1357 (70)	2005
F12	IB Monument	10,770	2.71	275 (31)	2007
F13	IB Commune d'Italie	8,129	6.18	136 (59)	2009
F14	IB Commune d'Allemagne	8,127	2.02	146 (33)	2006
F15	IB Personnalité du hockey sur glace	7,425	5.77	212 (54)	2008
F16	IB Cours d'eau	7,229	4.83	250 (52)	2006
F17	IB Club sportif ► IB Club de football	6,963	6.69	742 (78)	2006
F18	IB Livre	6,747	3.25	274 (33)	2006
F19	IB Édifice religieux	6,459	4.03	214 (36)	2007
F20	IB Commune d'Espagne ► IB Commune espagnole	6,259	4.73	184 (58)	2006
F21	IB Gare	5,959	7.75	285 (44)	2007
F22	IB Cycliste	5,943	7.11	105 (30)	2008
F23	IB Montagne	5,549	5.70	273 (47)	2006
F24	IB Voie parisienne	5,432	4.55	54 (26)	2006

F25	IB Écrivain	5,375	6.97	418 (33)	2007
F26	IB Personnalité militaire	5,368	6.78	366 (54)	2006
F27	IB Série télévisée	4,782	7.83	591 (31)	2007
F28	IBrélat catholique	4,748	1.53	202 (82)	2010
F29	IB Ville de Serbie	4,741	5.97	86 (67)	2007
F30	IB Athlète	4,739	2.92	174 (49)	2008
F31	IB Conflit militaire	4,616	10.31	386 (27)	2006
F32	IB Sportif ► IB Sportif, sportive	4,569	3.32	256 (51)	2007
F33	IB Personnage (fiction)	4,545	8.99	417 (51)	2007
F34	Chimiebox	4,494	9.29	699 (220)	2007
F35	IB Château	4,258	6.04	191 (39)	2007
F36	IB Canton de France	3,987	6.74	61 (17)	2006
F37	IB Ville des États-Unis	3,893	2.95	138 (42)	2009
F38	IB Parlementaire français	3,884	4.13	137 (14)	2007
F39	Album	3,879	6.65	601 (44)	2005
F40	IB Unité militaire	3,426	8.50	259 (32)	2007
F41	IB Rugbyman	3,329	5.77	164 (31)	2007
F42	IB Subdivision	3,315	2.68	143 (29)	2006
F43	IB Joueur de basket-ball	3,281	4.49	127 (38)	2007
F44	IB Ville de Chine	3,225	7.69	96 (39)	2007
F45	IB Île	3,218	5.66	214 (53)	2006
F46	IB Commune de Hongrie	3,181	4.73	95 (39)	2011
F47	IB Musique classique (personnalité)	3,111	5.91	261 (46)	2008
F48	IB Commune de Slovaquie ► IB Village et ville de Slovaquie	2,949	3.76	79 (32)	2009
F49	IB Localité des Pays-Bas	2,862	2.16	51 (40)	2010
F50	IB Commune de Suisse	2,792	11.06	240 (60)	2006

Appendix B

Value Comparison

Contents

In this section we describe further details of the value comparison used in the edit detection (See Section 3.2.1). We first discuss technical aspects of comparing atomic values. We then describe how array value types can be compared.

B.1 Distance Functions for Atomic Values

We differentiate between the following data types: personal name, number, date, url and enum. enum refers to a data type where we only differentiate between identical/not identical values. This is necessary if two values have no continuous similarity like an md5-hash string.

For each type, we predefine a suitable set of string distance functions. As described in Section 3.2.1, we mostly use well known distance functions for which fast and reliable implementations are available. We will now give a short description for each function we use. Most functions are similarity functions with results $r \in [0, 1]$. To use these functions in our framework, we return $1 - r$ as distance value. Some functions have preconditions which must be met by the values. These can be technical limitations (e.g., length of values) or restrictions on the content of the value (e.g., value is a number).

Jaccard See Section 3.2.1.
Preconditions: None.

Levenshtein See Section 3.2.1.
Preconditions: Maximum text length < 100 characters.

Monge-Elkan The Monge-Elkan similarity [ME97] is defined by splitting the input strings into tokens and summing up the distance of the best matches. The values are normalized to a range $[0, 1]$. Our implementation uses scaled Levenshtein similarity to compare the individual tokens. This metric is a combination of Levenshtein and Jaccard distance.
Preconditions: None (precondition for individual token pairs same as for Levenshtein)
The distance is defined as 1 - (Monge-Elkan similarity)

Jaro-Winkler The Jaro-Winkler distance [Win99] was specifically developed for comparing person names. The basic algorithm is similar to Levenshtein but it favors words that start with the same prefix. This is useful to detect abbreviations of a name.
Preconditions: None.

Encoding Transforms input strings by removing diacritics and normalizing composite characters and ligatures. This distance function detects pairs of strings which only differ in encoding.
Precondition: None
Let a' and b' be the transformed strings. We define:

$$\text{encoding}(a, b) := \begin{cases} 0 & \text{if } a' = b' \\ 1 & \text{if } a' \neq b' \end{cases}$$

URL A specific string distance for URLs. Extracts *host* and *path* from the input string. Ignores information such as port number, protocol and query strings. E.g. from `http://test.domain.org/path?q=query` date host=`test.domain` and path=`path`. The similarity uses only these two values. The idea is that host and path are more stable than other parts of the URL which might change over time. We consider the host to be more stable than the path.
Preconditions: a and b have URL format and host and path (h_a, p_a), (h_b, p_b) can be extracted from them. For given $\alpha, \beta \in [0, 1]$, we define

$$\text{URL}(a, b) := \begin{cases} \alpha & \text{if } h_a = h_b \wedge p_a \neq p_b \\ \beta & \text{if } h_a \neq h_b \wedge p_a = p_b \\ 0 & \text{if } h_a = h_b \wedge p_a = p_b \\ 1 & \text{else} \end{cases}$$

For our experiments, we used $\alpha = 0.1$ and $\beta = 0.9$.

Number Compares two numbers based on the relative distance between them.
Precondition: a and b can be parsed to numbers that represent unix time stamps. Numbers are positive.
Let n_a be the number parsed from a and n_b be the number parsed from b. Let $n_a \leq n_b$ Then:

$$\text{number}(a, b) = 1 - (n_a / n_b)$$

Date Compares dates by extracting Unix-timestamp and applying number distance on it. The date distance requires a predefined maximal distance δ that represents the upper bound for the permitted distance in seconds.
Precondition: a and b can be parsed to numbers. Dates are from current Unix epoch (after 01-01-1970)
With n_a and n_b as above, and a given δ, we define:

$$\text{date}(a,b) := \begin{cases} 1 & \text{if } n_b - n_a > \delta \\ \dfrac{(n_b - n_a)}{\delta} & \text{else} \end{cases}$$

Enum A very simple similarity function.
Precondition: None.
Enum distance is defined as.

$$\text{enum}(a,b) := \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

B.2 Value Similarity for Arrays

In Section 3.2.1, we discussed a string distance function for atomic value domains. In the data sets which we consider in this work, atomic domains are dominant. This is in part related to the modeling ambiguity we discussed in Section 2.5. However, array type domains exist. In this section, we discuss how the atomic distance can be extended to array domain types.

For array type values, we must consider addition/removal of values and the permutation of the values. Consider the following three values:

> ***Example B.1:*** Array type values.
>
> v_1 [Adam, Bob, Charles]
>
> v_2 [Bob, Charles, Adam]
>
> v_3 [Adam, Bart, Charles]

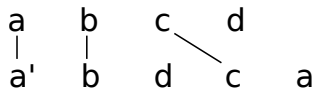
v_1 and v_2 contain the same components but in different order. In fact, each element is at another position. v_1 and v_3 share only two atomic values. However, these values are at the same position. Additionally, the third value pair (Bob and Bart) shows some similarities. Whether v_2 or v_3 is more similar to v_1 depends on the properties of the key. For example, assume that the arrays represent the authors of scientific publications. The order of authors is very important here as the first author is often regarded as the primary contributor to the work. In this case, we might prefer v_3 as match for v_1 with the assumption that the atomic value *Bob* was renamed to *Bart*. In a scenario where there is no order of the atomic values, we would prefer v_2.

> ***Observation 1:*** For some keys with array values, the order of the values is important.

For some keys, the atomic values in an array value are replaced frequently. These are often administrative metadata. For example, the record of a book might store IDs of persons waiting to lend the book. For these data elements, we can observe modifications where atomic values (references to waiting patrons) are added and modifications where they are deleted again. Some of the data sets we consider in this work have data elements that track the time of modifications. Every time a data set is modified, a new atomic value (some type of timestamp) is added to the value of the element.

> ***Observation 2:*** For some data elements, adding and deleting atomic values is normal. Values that differ in this way should be considered to be similar.

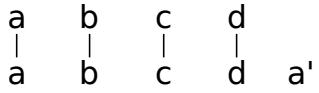
Assume we want to determine the distance of the value arrays $[a, b, c, d]$ and $[a', b, d, c, a]$. According to Observation 1, we might need to consider the order of elements. We can do that by computing the longest common subsequence (lcs) of the arrays. We use a modified lcs algorithm that considers value pairs with a distance $< \delta$ for a given δ. If a and a' are sufficiently similar, we obtain a mapping.



Let $c := (a_1, b_1), \ldots, (a_k, b_k)$ be the mapping pairs of the lcs. We can compute the similarity of the lcs as:

$$sim_{lcs} = \frac{\sum_{1 \leq i \leq k} sim(a_i, b_i)}{k}$$

However, there might be situations where preserving the order is less relevant. Consider this lcs in case we reordered the second array:



The lcs is not only longer, it also has a smaller distance, as all elements are pairwise identical. The degree of an order change can be computed by counting the number of inversions of the lcs. Let i be the position of element a and j be the position of an element b in the original value. Let $p(i)$ and $p(j)$ be the positions of these elements in the permuted array. If $i < j$ and $p(i) > p(j)$, we call (a, b) inverted. The inversion

count of $[a', b, d, c, a]$ is 4 (a-b, a-c, a-d and c-d). The maximal number of inversions in an lcs is $|lcs| \cdot (|lcs| - 1)$. With this, we can define a similarity for the longest common subsequence.

Definition B.1 (Core Similarity): For a longest common subsequence lcs with sim_{lcs} and inversion count inv let $\alpha \in [0, 1]$. We call

$$sim_{core} := \alpha \cdot sim_{lcs} + (1 - \alpha) \cdot \frac{inv}{|lcs| \cdot (|lcs| - 1)}$$

the **core similarity** of lcs.

α is used to determine how important the order of data elements is. For $\alpha = 1$, we ignore order changes completely. The core similarity only considers the longest common subsequence. Assume that we want to ignore the order of values. In this case, we can pick a permutation for which the lcs contains zero values or only a single value (depending on the pairwise similarities). This permutation would produce the best core similarity. Obviously, we must favor longer subsequences. At the same time, we must consider the fact that some elements grow and shrink naturally. We define:

Definition B.2 (Array Similarity): Let sim_{core} be a core similarity of arrays a and b as described above. Let $\beta \in [0, 1]$. We call

$$sim_{array} := \beta \cdot sim_{core} + (1 - \beta) \cdot \frac{2 \cdot |lcs|}{|a| + |b|}$$

The **array similarity** of a and b.

β determines the importance of added or deleted elements as well as changes of size. If $\beta = 1$, only the core similarity will be considered. This is useful for array values that tend to change by appending values to the existing ones. For example, a data element might track the dates of last ten loans of a book. A new loan adds a value to the array and removes the oldest one (if there are ten entries already).

To determine the array similarity of array a and b, we need to compare the longest common sub-sequences for all permutations of b. For an array of n elements, the number of permutations is bounded from above by $n!$ (if all elements of the array are pairwise distinguishable). This is only feasible if the arrays are short. For longer arrays, we use a greedy algorithm that groups together atomic values with decreasing similarity.

Appendix C

Test Collections

Contents

This chapter discusses format specifications of the test collections that we provide as part of this work. There are two types of test collections:

- The case-based test collection (see Section 4.3.3)
- The embedded test collection (See Section 4.3.4)

We will now discuss the format and the available data sets for both types.

Both test collections have been published on `https://zenodo.org` under the Open Data Commons Attribution License (ODC-BY). The datasets can be found at [Rei18].

C.1 Case-based Test Collection

The case-based test collection consists of isolated test cases directly derived from corrections to DBLP. The test cases are observed between June 1999 and October 2015. See Section 4.3.3 for details on the creation of the collection.

Table C.1: Node properties for publications.

title (Coverage: 100%) The title of the publication. Might contain markup such as latex commands. Very few records have multiple titles in different languages. These titles are merged into a single string.
year (∼100%) The year in which the document was published.
url (99.6%) An internal url used by DBLP to locate the venue this paper belongs to. The url can be more detailed than the venue information. For example, it might tell on which workshop of a conference the paper was published. However, this information is weakly formatted and difficult to parse.
ee (93.6%) An external reference, often to the webpage of the publisher.
pages (91.6%) The pagination information for the document. Very different formats are in use such as 15-20 or 25:1-25:10.
booktitle (88.5%) The title of a proceeding or the name of a journal. Usage similar to url.
volume (43.9%) The volume of a journal.
number (36.8%) The number of a journal issue.

C.1.1 Format

For each isolated test case, a pair of XML files is provided. One file contains the local state of the metadata before the correction. The other file contains the local state after the correction. The files are names with the number of the observation that defined its state. E.g., a test case might consist of files 1000.xml and 1001.xml. In this case, 1000.xml contains the most recent observation of the defect and 1001.xml the state directly after the correction. The metadata is provided as a local graph. See Section 4.3.3 for details.

The format of this test collection is defined in an XML-Schema definition. The schema mainly consists of nodes and edges between them.

A node is defined as:

```
<xs:complexType name="nodeType">
    <xs:sequence>
        <xs:element name="property" type="propertyType" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence>

    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
```

where

- attribute **label** is the type of the node, e.g., DOCUMENT.

- attribute **id** is a unique identifier (enforced by the XML-Schema definition) for the node.
- child element **property** can hold arbitrary key-value pairs.

For properties, we provide the most frequent values found in DBLP records. These values are described in Table C.1.

Example for a node:

```
<node label="DOCUMENT" id="conf/icchp/NeverydB94">
    <property key="venue" value="conf/icchp"/>
    <property key="year" value="1994"/>
    <property key="booktitle" value="ICCHP"/>
    <property key="title" value="The Ultrasonic Navigating Robot, WALKY."/>
</node>
```

Edges are defined as:

```
<xs:complexType name="edgeType">
    <xs:sequence>
        <xs:element name="property" type="propertyType" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="source" type="xs:string" use="required"/>
    <xs:attribute name="target" type="xs:string" use="required"/>
    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="weight" type="xs:double" use="required"/>
    <xs:attribute name="directed" type="xs:boolean" use="required"/>
</xs:complexType>
```

where **property** as in Table C.1 and

- attribute **source** references id of source node.
- attribute **target** references id of target node.
- attribute **label** is the type of the relation, e.g., CREATED.
- attribute **weight** stores weight of the edge.
- attribute **directed** is true if edge is directed and false if not.

Example of an edge:

```
<edge source="homepages/89/5765" target="homepages/12/5446"
    label="CO_CREATED" weight="1.0" directed="false"/>
```

The test case document starts with an element that provides the type of the correction (e.g., MERGE) and the identifiers of the primary nodes.

The full XML-Schema definition is printed below:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

  <xs:element name="graph">

    <xs:complexType>
      <xs:sequence>
        <xs:element name="metadata" type="metadataType"/>

          <xs:element name="nodes">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="node" type="nodeType"
                    minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
              <xs:attribute name="count" type="xs:int" use="required"/>
            </xs:complexType>
          </xs:element>

          <xs:element name="edges">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="edge" type="edgeType"
                    minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
              <xs:attribute name="count" type="xs:int" use="required"/>
            </xs:complexType>
          </xs:element>

      </xs:sequence>
      <xs:attribute name="date" type="xs:string" use="required"/>
    </xs:complexType>

    <xs:key name="nodeId">
      <xs:selector xpath="nodes/node"></xs:selector>
      <xs:field xpath="@id"/>
    </xs:key>

    <xs:keyref refer="nodeId" name="edgeSourceRef">
      <xs:selector xpath="edges/edge"></xs:selector>
      <xs:field xpath="@source"></xs:field>
    </xs:keyref>

    <xs:keyref refer="nodeId" name="edgeTargetRef">
      <xs:selector xpath="edges/edge"></xs:selector>
      <xs:field xpath="@target"></xs:field>
```

```
    </xs:keyref>

</xs:element>

<xs:complexType name="metadataType">
    <xs:sequence>
        <xs:element name="changetype">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="MERGE"/>
                    <xs:enumeration value="MERGE_RENAME"/>
                    <xs:enumeration value="SPLIT"/>
                    <xs:enumeration value="SPLIT_RENAME"/>
                    <xs:enumeration value="DISTRIBUTE"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>

        <xs:element name="before">
            <xs:complexType>
                <xs:sequence minOccurs="1" maxOccurs="unbounded">
                    <xs:element name="node" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>

        <xs:element name="after">
            <xs:complexType>
                <xs:sequence minOccurs="1" maxOccurs="unbounded">
                    <xs:element name="node" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="propertyType">
    <xs:attribute name="key" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="nodeType">
    <xs:sequence>
        <xs:element name="property" type="propertyType" minOccurs="0"
            maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:string" use="required"/>
```

```
  </xs:complexType>

  <xs:complexType name="edgeType">
    <xs:sequence>
      <xs:element name="property" type="propertyType" minOccurs="0"
          maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="source" type="xs:string" use="required"/>
    <xs:attribute name="target" type="xs:string" use="required"/>
    <xs:attribute name="label" type="xs:string" use="required"/>
    <xs:attribute name="weight" type="xs:double" use="required"/>
    <xs:attribute name="directed" type="xs:boolean" use="required"/>
  </xs:complexType>
</xs:schema>
```

The following listing contains an example of a very simple merge test case. In the example, the nodes with identifiers *homepages/58/7937* and *homepages/89/5765* are merged.

```
<graph date="2002-12-13 00:00:00">
  <metadata>
    <changetype>MERGE</changetype>
    <before>
      <node>homepages/58/7937</node>
      <node>homepages/89/5765</node>
    </before>
    <after>
      <node>homepages/89/5765</node>
    </after>
  </metadata>
  <nodes count="9">
    <node label="DOCUMENT" id="conf/icchp/NeverydB94">
      <property key="venue" value="conf/icchp"/>
      <property key="year" value="1994"/>
      <property key="booktitle" value="ICCHP"/>
      <property key="title" value="The Ultrasonic Navigating Robot,
          WALKY."/>
    </node>
    <node label="DOCUMENT" id="conf/wsc/RandellHB99">
      <property key="venue" value="conf/wsc"/>
      <property key="year" value="1999"/>
      <property key="booktitle" value="Winter Simulation Conference"/>
      <property key="title" value="Incremental system development of large
          discrete-event simulation models."/>
    </node>
    <node label="PERSON" id="homepages/12/5446"/>
    <node label="PERSON" id="homepages/16/5747"/>
    <node label="PERSON" id="homepages/17/5330"/>
```

```
    <node label="PERSON" id="homepages/58/7937"/>
    <node label="PERSON" id="homepages/89/5765"/>
    <node label="VENUE" id="conf/icchp"/>
    <node label="VENUE" id="conf/wsc"/>
  </nodes>
  <edges count="14">
    <edge source="homepages/17/5330" target="homepages/12/5446"
        label="CO_CREATED" weight="1.0" directed="false"/>
    <edge source="homepages/58/7937" target="homepages/16/5747"
        label="CO_CREATED" weight="1.0" directed="false"/>
    <edge source="homepages/89/5765" target="homepages/12/5446"
        label="CO_CREATED" weight="1.0" directed="false"/>
    <edge source="homepages/89/5765" target="homepages/17/5330"
        label="CO_CREATED" weight="1.0" directed="false"/>
    <edge source="homepages/12/5446" target="conf/wsc/RandellHB99"
        label="CREATED" weight="1.0" directed="true">
      <property key="name" value="Lars G. Holst"/>
      <property key="position" value="2"/>
    </edge>
    <edge source="homepages/16/5747" target="conf/icchp/NeverydB94"
        label="CREATED" weight="1.0" directed="true">
      <property key="name" value="Hakan Neveryd"/>
      <property key="position" value="1"/>
    </edge>
    <edge source="homepages/17/5330" target="conf/wsc/RandellHB99"
        label="CREATED" weight="1.0" directed="true">
      <property key="name" value="Lars G. Randell"/>
      <property key="position" value="1"/>
    </edge>
    <edge source="homepages/58/7937" target="conf/icchp/NeverydB94"
        label="CREATED" weight="1.0" directed="true">
      <property key="name" value="Gunnar Bolmsjö"/>
      <property key="position" value="2"/>
    </edge>
    <edge source="homepages/89/5765" target="conf/wsc/RandellHB99"
        label="CREATED" weight="1.0" directed="true">
      <property key="name" value="Gunnar S. Bolmsjö"/>
      <property key="position" value="3"/>
    </edge>
    <edge source="homepages/12/5446" target="conf/wsc"
        label="CREATED_AT" weight="1.0" directed="true"/>
    <edge source="homepages/16/5747" target="conf/icchp"
        label="CREATED_AT" weight="1.0" directed="true"/>
    ...
  </edges>
</graph>
```

C.1.2 Data Sets

The following data sets are provided:

- **case-based-all**: collection consisting of all case-based test cases.
- **case-based-big-{all,coauthor}-{5,10,50}**: collection consisting of test cases where each primary node provides a minimal amount of local information.

The size restricted collections might be used to evaluate algorithms which require a significant amount of local information. See Section 4.5.1 for details. E.g. **case-based-big-coauthor-10** contains only cases where each primary node has at least 10 coauthors. See Table 4.10 for the size of the different collections.

C.2 Embedded Test Collection

The embedded test collection consists of full copies of DBLP and files which contain lists of known ER related defects in these copies. For details, see Section 4.3.4.

The test collection contains the following full copies of the DBLP collection:

- dblp2013.xml.gz
- dblp2014.xml.gz
- dblp2015.xml.gz
- dblp2016.xml.gz
- dblp2017.xml.gz (for comparison)

A description on the content of these files can be found on[1].

The embedded collection is created by comparing two states of DBLP from different years. We compare the DBLP state at the beginning of 2013, 2014, 2015 and 2016 with the state at the beginning of 2017. For each pair of states, we provide lists of defects which are contained in the earlier state but fixed in the newer state. We provide two types of lists:

- short: a list that contains the defects and the affected DBLP author profiles.
- full: a list that contains the defects and the affected author profiles as well as the signatures which were assigned to them in both states.

[1] `http://dblp.dagstuhl.de/faq/What+do+I+find+in+dblp+xml.html`

The base format of a defect is:

```
<defect type="Split" number="x">
  <source>
    <profile authorid="homepages/97/1065" />
  </source>
  <target>
    <profile authorid="homepages/164/7743" />
    <profile authorid="homepages/97/1065" />
  </target>
</defect>
```

In the example, the signatures from author profile homepages/97/1065 are redistributed to homepages/164/7743 and homepages/97/1065. In the full files, profiles have signatures attached to them:

```
<defect type="Split" number="x">
  <source>
    <profile authorid="homepages/97/1065">
      <signature pkey="journals/cor/MarcotteMS95" pos="1"
        surface="Gerald P. Marquis"/>
      <signature pkey="journals/infsof/Marquis02" pos="0"
        surface="Gerald P. Marquis"/>
      </profile>
    </source>
    <target>
      <profile authorid="homepages/164/7743">
        <signature pkey="journals/cor/MarcotteMS95" pos="1"
          surface="Gerald Marquis"/>
      </profile>
      <profile authorid="homepages/97/1065">
        <signature pkey="journals/infsof/Marquis02" pos="0"
          surface="Gerald P. Marquis"/>
      </profile>
    </target>
  </defect>
```

A signature consists of three components:

- pkey: the publication key in DBLP.
- pos: the position of the signature in the author list, starting at 0.
- surface: the surface form of the signature.

Short and full lists are defined by the following XML-Schema definition:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

  <xs:element name="embedded">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="defect">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="source">
                <xs:complexType>
                  <xs:sequence maxOccurs="unbounded">
                    <xs:element name="profile" type="profileType"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="target">
                <xs:complexType>
                  <xs:sequence maxOccurs="unbounded">
                    <xs:element name="profile" type="profileType"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="type">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="Split"/>
                  <xs:enumeration value="Merge"/>
                  <xs:enumeration value="Distribute"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="number" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="profileType">
    <xs:sequence>
      <xs:element name="signature" minOccurs="0"
          maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="pkey" type="xs:string"/>
```

```
            <xs:attribute name="pos" type="xs:integer"/>
            <xs:attribute name="surface" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="authorid" type="xs:string"/>
    </xs:complexType>

</xs:schema>
```

The files which are part of the test collection are named as follow:

- **full-yyyy-2017.xml** the defect list with all signatures. yyyy $\in$ $\{2013, 2014, 2015, 2016\}$: year of first observation.
- **short-yyyy-2017.xml** the defect list without signatures. yyyy $\in$ $\{2013, 2014, 2015, 2016\}$: year of first observation.
- **dblpyyyy.xml** full version of the DBLP data set at the beginning of yyyy $\in$ $\{2013, 2014, 2015, 2016, 2017\}$(see above).

I.e., **full-2013-2017.xml** contains the defects which we observed by comparing the states of 2013 and 2017. The number of defects in the individual lists can be found in Table 4.5.

Index

Bibliography

[AAoLP13] The American Library Association, The Canadian Library Association, CILIP: Chartered Institute of Library, and Information Professionals. The RDA: Resource Description and Access, 2013. `http://http://rda-jsc.org/` (accessed June 2017).

[AKM08] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. What's a typical commit? A characterization of open source software repositories. In *ICPC '08: Int. Conf. on Program Comprehension, Amsterdam, The Netherlands*, pages 182–191. IEEE, 2008. doi:10.1109/ICPC.2008.24.

[Ame72] American Library Association. *The National union catalog, pre-1956 imprints*. Mansell, London, UK, 1972.

[BBMU11] Nedyalko Borisov, Shivnath Babu, NagaPramod Mandagere, and Sandeep Uttamchandani. Warding off the dangers of data corruption with amulet. In *SIGMOD '11: Proc. of the ACM Int. Conf. on Management of Data, Athens, Greece*, pages 277–288. ACM, 2011. doi:10.1145/1989323.1989353.

[BG06] Indrajit Bhattacharya and Lise Getoor. A Latent Dirichlet Model for Unsupervised Entity Resolution. In *Proc. of the Sixth SIAM Int. Conf. on Data Mining, Bethesda, MD, USA*, pages 47–58. SIAM, 2006. doi:10.1137/1.9781611972764.5.

[BG07] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007. doi:10.1145/1217299.1217304.

[BKM06] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive Blocking: Learning to Scale Up Record Linkage. In *ICDM '06: Proc. of the 6th IEEE Int. Conf. on Data Mining, Hong Kong, China*, pages 87–96. IEEE Computer Society, 2006. doi:10.1109/ICC.2007.399.

[BMC+03] Mikhail Bilenko, Raymond J. Mooney, William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. Adaptive Name Matching in

Information Integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003. doi:10.1109/MIS.2003.1234765.

[BND+11] Kevin W. Boyack, David Newman, Russell J. Duhon, Richard Klavans, Michael Patek, Joseph R. Biberstine, Bob Schijvenaars, André Skupin, Nianli Ma, and Katy Börner. Clustering more than two million biomedical publications: Comparing the accuracies of nine text-based similarity approaches. *PloS one*, 6(3):e18029, 2011. doi:10.1371/journal.pone.0018029.

[Bov06] John Bovey. Adding User-Editing to a Catalogue of Cartoon Drawings. In *ECDL '06: Research and Advanced Technology for Digital Libraries, Proc., Alicante, Spain*, volume 4172 of *LNCS*, pages 457–460. Springer, 2006. doi:10.1007/11863878_42.

[Bra03] Karl Branting. A Comparative Evaluation of Name-Matching Algorithms. In *ICAIL '03: Proc. of the 9th Int. Conf. on Artificial Intelligence and Law, Edinburgh, UK*, pages 224–232. ACM, 2003. doi:10.1145/1047788.1047837.

[CCG+14] David Carmel, Ming-Wei Chang, Evgeniy Gabrilovich, Bo-June Paul Hsu, and Kuansan Wang. ERD'14: Entity Recognition and Disambiguation Challenge. *SIGIR Forum*, 48(2):63–77, 2014. doi:10.1145/2701583.2701591.

[CFN+10] Ricardo G. Cota, Anderson A. Ferreira, Cristiano Nascimento, Marcos André Gonçalves, and Alberto H. F. Laender. An unsupervised heuristic-based hierarchical method for name disambiguation in bibliographic citations. *JASIST*, 61(9):1853–1870, 2010. doi:10.1002/asi.21363.

[Che13] Wenlong Chen. OAI-Metadata Harvester für historische Daten. Master's thesis, University of Trier, Germany, 2013.

[CJ06] Gregory R. Crane and Alison Jones. The challenge of virginia banks: an evaluation of named entity analysis in a 19th-century newspaper collection. In *JCDL '06: Proc. of the Joint Conf. on Digital Libraries, Chapel Hill, NC, USA*, pages 31–40. ACM, 2006. doi:10.1145/1141753.1141759.

[CKH+07] Aron Culotta, Pallika Kanani, Robert Hall, Michael Wick, and Andrew McCallum. Author disambiguation using error-driven machine learning with a ranking loss function. In *IIWeb-07: Proc. of the 6th Int. Workshop on Information Integration on the Web, Vancouver, Canada*, 2007.

[CKT06] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *SIGKDD '06: Proc. of the 12th Int. Conf. on*

Knowledge Discovery and Data Mining, Philadelphia, PA, USA, pages 554–560. ACM, 2006. doi:10.1145/1150402.1150467.

[CRF03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IIWeb-03: Proc. of IJCAI-03 Workshop on Information Integration on the Web, Acapulco, Mexico*, pages 73–78, 2003.

[DDC13] Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *VLDB J.*, 22(5):665–687, 2013. doi:10.1007/s00778-013-0324-z.

[DGA11] Ciriaco Andrea D'Angelo, Cristiano Giuffrida, and Giovanni Abramo. A heuristic approach to author name disambiguation in bibliometrics databases for large-scale research assessments. *JASIST*, 62(2):257–269, 2011. doi:10.1002/asi.21460.

[DKL08] Hongbo Deng, Irwin King, and Michael R. Lyu. Formal Models for Expert Finding on DBLP Bibliography Data. In *ICDM '08: Proc. of the 8th Int. Conf. on Data Mining*, pages 163–172. IEEE Computer Society, 2008. doi:10.1109/ICDM.2008.29.

[ECdSJ10] Luiz Osvaldo Evangelista, Eli Cortez, Altigran Soares da Silva, and Wagner Meira Jr. Adaptive and Flexible Blocking for Record Linkage Tasks. *JIDM*, 1(2):167–182, 2010.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007. doi:10.1109/TKDE.2007.250581.

[EL05] Ergin Elmacioglu and Dongwon Lee. On six degrees of separation in DBLP-DB and more. *SIGMOD Record*, 34(2):33–40, 2005. doi:10.1145/1083784.1083791.

[Ell10] Sarah Elliot. Survey of author name disambiguation: 2004 to 2010. 2010. obtained from `http://digitalcommons.unl.edu/libphilprac/473` (accessed October 2016).

[Ens09] Martin Enserink. Are You Ready to Become a Number? *Science*, 323(5922):1662–1664, 2009. doi:10.1126/science.323.5922.1662.

[FAW08] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Analysing Bug Prediction Capabilities of Static Code Metrics in Open Source Software. In *IWSM '08, Metrikon 2008, and Mensura 2008: Software Process and Product Measurement, Munich, Germany*, volume 5338 of *LNCS*, pages 331–343. Springer, 2008. doi:10.1007/978-3-540-89403-2_27.

[FGA+12] Anderson A. Ferreira, Marcos André Gonçalves, Jussara M. Almeida, Alberto H. F. Laender, and Adriano Veloso. A tool for generating synthetic authorship records for evaluating author name disambiguation methods. *Inf. Sci.*, 206:42–62, 2012. doi:10.1016/j.ins.2012.04.022.

[FGL12] Anderson A. Ferreira, Marcos André Gonçalves, and Alberto H. F. Laender. A brief survey of automatic methods for author name disambiguation. *ACM Sigmod Record*, 41(2):15–26, 2012. doi:10.1145/2350036.2350040.

[FJ13] Katrina Fenlon and Virgil E. Varvel Jr. Local histories in global digital libraries: identifying demand and evaluating coverage. In *JCDL '13: 13th Joint Conf. on Digital Libraries, Indianapolis, IN, USA*, pages 191–194. ACM, 2013. doi:10.1145/2467696.2467742.

[FK13] Daniel Fried and Stephen G. Kobourov. Maps of Computer Science. *arXiv (CoRR)*, 1304.2681, 2013. arXiv:1304.2681.

[FR08] M. Foulonneau and J. Riley. *Metadata for Digital Resources: Implementation, Systems Design and Interoperability*. Chandos Information Professional Series. Chandos Publishing, 2008.

[Fre79] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978-1979. doi:10.1016/0378-8733(78)90021-7.

[FVGL10] Anderson A. Ferreira, Adriano Veloso, Marcos André Gonçalves, and Alberto H. F. Laender. Effective self-training author name disambiguation in scholarly digital libraries. In *JCDL '10: Proc. of the 2010 Joint Int. Conf. on Digital Libraries, Gold Coast, Queensland, Australia*, pages 39–48. ACM, 2010. doi:10.1145/1816123.1816130.

[FVGL14] Anderson A. Ferreira, Adriano Veloso, Marcos André Gonçalves, and Alberto H. F. Laender. Self-training author name disambiguation for information scarce scenarios. *JASIST*, 65(6):1257–1278, 2014. doi:10.1002/asi.22992.

[FWP+11] Xiaoming Fan, Jianyong Wang, Xu Pu, Lizhu Zhou, and Bing Lv. On Graph-Based Name Disambiguation. *J. Data and Information Quality*, 2(2):10, 2011. doi:10.1145/1891879.1891883.

[GI95] Jerrold W. Grossman and Patrick D. F. Ion. On a portion of the well-known collaboration graph. *Congressus Numerantium*, 108:129–132, 1995.

[GN02] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proc. of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, 2002. doi:10.1073/pnas.122653799.

[GW78] Michael Gorman and Paul Walter Winkler. *Anglo-American cataloguing rules*. Library Association Publishing, 1978.

[GWO09] Jun Gong, Lidan Wang, and Douglas W. Oard. Matching person names through name transformation. In *CIKM '09: Proc. of the 18th Conf. on Information and Knowledge Management, Hong Kong, China*, pages 1875–1878. ACM, 2009. doi:10.1145/1645953.1646253.

[HEG06] Jian Huang, Seyda Ertekin, and C. Lee Giles. Efficient Name Disambiguation for Large-Scale Databases. In *PKDD '06: 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases, Berlin, Germany*, volume 4213 of *LNCS*, pages 536–544. Springer, 2006. doi:10.1007/11871637_53.

[HGZ+04] Hui Han, C. Lee Giles, Hongyuan Zha, Cheng Li, and Kostas Tsioutsiouliklis. Two supervised learning approaches for name disambiguation in author citations. In *JCDL '04: Proc. of the Joint Conf. on Digital Libraries, Tucson, AZ, USA*, pages 296–305. ACM, 2004. doi:10.1145/996350.996419.

[Hic12] Diana Hicks. Performance-based university research funding systems. *Research policy*, 41(2):251–261, 2012. doi:10.1016/j.respol.2011.09.007.

[Hir05] Jorge E Hirsch. An index to quantify an individual's scientific research output. *Proc. of the National academy of Sciences of the United States of America*, 102(46):16569–16572, 2005.

[HPM14] Cornelia Hedeler, Bijan Parsia, and Brigitte Mathiak. Using the semantic web for author disambiguation - are we there yet? In *ISWC '14: Proc. of the 13th Int. Semantic Web Conf., Riva del Garda, Italy*, volume 1272 of *CEUR Workshop Proc.*, pages 449–452. CEUR-WS.org, 2014.

[HSW10] Monika Henzinger, Jacob Suñol, and Ingmar Weber. The stability of the h-index. *Scientometrics*, 84(2):465–479, 2010. doi:10.1007/s11192-009-0098-7.

[HXZG05] Hui Han, Wei Xu, Hongyuan Zha, and C. Lee Giles. A hierarchical naive Bayes mixture model for name disambiguation in author citations. In *SAC '05: Proc. of the Symposium on Applied Computing, Santa Fe, NM, USA*, pages 1065–1069. ACM, 2005. doi:10.1145/1066677.1066920.

[HYM+16] Hussein Hazimeh, Iman Youness, Jawad Makki, Hassan Noureddine, Julien Tscherrig, Elena Mugellini, and Omar Abou Khaled. Leveraging Co-authorship and Biographical Information for Author Ambiguity Resolution in DBLP. In *AINA '16: 30th Int. Conf. on Advanced Information Networking and Applications, Crans-Montana, Switzerland*, pages 1080–1084. IEEE Computer Society, 2016. doi:10.1109/AINA.2016.61.

[HYQQ09] Zhixing Huang, Yan Yan, Yuhui Qiu, and Shuqiong Qiao. Exploring Emergent Semantic Communities from DBLP Bibliography Database. In *ASONAM '09: Int. Conf. on Advances in Social Network Analysis and Mining, Athens, Greece*, pages 219–224. IEEE Computer Society, 2009. doi:10.1109/ASONAM.2009.6.

[HZ09] Xianpei Han and Jun Zhao. Named entity disambiguation by leveraging wikipedia semantic knowledge. In *CIKM '09: Proc. of the 18th Conf. on Information and Knowledge Management, Hong Kong, China*, pages 215–224. ACM, 2009. doi:10.1145/1645953.1645983.

[HZG05] Hui Han, Hongyuan Zha, and C. Lee Giles. Name disambiguation in author citations using a K-way spectral clustering method. In *JCDL '05: Proc. of the Joint Conf. on Digital Libraries, Denver, CO, USA*, pages 334–343. ACM, 2005. doi:10.1145/1065385.1065462.

[IFL98] IFLA Study Group on the Functional Requirements for Bibliographic Records and International Federation of Library Associations and Institutions. *Functional requirements for bibliographic records: final report.* UBCIM publications. K.G. Saur, 1998.

[Joc99] Uwe Jochum. *Kleine Bibliotheksgeschichte, 2nd Edition.* Reclams Universal-Bibliothek. P. Reclam, Stuttgart, Germany, 1999.

[KBM+07] Nishikant Kapoor, John T. Butler, Sean M. McNee, Gary C. Fouty, James A. Stemper, and Joseph A. Konstan. A Study of Citations in Users' Online Personal Collections. In *ECDL '07: 11th European Conf. on Research and Advanced Technology for Digital Libraries, Budapest, Hungary*, volume 4675 of *LNCS*, pages 404–415. Springer, 2007. doi:10.1007/978-3-540-74851-9_34.

[Kir14] Thomas Kirsch. Value Type Identification and Classification of Modifications in Digital Libraries Metadata. Master's thesis, University of Trier, Germany, 2014.

[KKL+11] In-Su Kang, Pyung Kim, Seungwoo Lee, Hanmin Jung, and Beom-Jong You. Construction of a large-scale test set for author disambiguation. *Inf. Process. Manage.*, 47(3):452–465, 2011. doi:10.1016/j.ipm.2010.10.001.

[KMP07] Pallika H. Kanani, Andrew McCallum, and Chris Pal. Improving Author Coreference by Resource-Bounded Information Gathering from the Web. In Manuela M. Veloso, editor, *IJCAI '07: Proc. of the 20th Int. Joint Conf. on Artificial Intelligence, Hyderabad, India*, pages 429–434, 2007.

[KNL+09] In-Su Kang, Seung-Hoon Na, Seungwoo Lee, Hanmin Jung, Pyung Kim, Won-Kyung Sung, and Jong-Hyeok Lee. On co-authorship for

author disambiguation. *Inf. Process. Manage.*, 45(1):84–97, 2009. doi:10.1016/j.ipm.2008.06.006.

[LASM16] Gilles Louppe, Hussein T. Al-Natsheh, Mateusz Susik, and Eamonn James Maguire. Ethnicity Sensitive Author Disambiguation Using Semi-supervised Learning. In *KESW '16: Proc. of the 7th Int. Conf. on Knowledge Engineering and Semantic Web, Prague, Czech Republic*, volume 649 of *Comm. in Computer and Information Science*, pages 272–287. Springer, 2016. doi:10.1007/978-3-319-45880-9_21.

[LdLM+08] Alberto H. F. Laender, Carlos José Pereira de Lucena, José Carlos Maldonado, Edmundo de Souza e Silva, and Nivio Ziviani. Assessing the research and education quality of the top Brazilian Computer Science graduate programs. *SIGCSE Bulletin*, 40(2):135–145, 2008. doi:10.1145/1383602.1383654.

[Lev66] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals (translation from russian). *Soviet Physics Doklady*, 10(8):707–710, 1966.

[Ley09] Michael Ley. DBLP - Some Lessons Learned. *PVLDB*, 2(2):1493–1500, 2009. doi:10.14778/1687553.1687577.

[LH10] Felipe Hoppe Levin and Carlos A. Heuser. Evaluating the Use of Social Networks in Author Name Disambiguation in Digital Libraries. *JIDM*, 1(2):183–198, 2010.

[Lib86] Library of Congress. Library of Congress: Subject headings (Mikrofiche), 1986.

[LIDK+14] Wanli Liu, Rezarta Islamaj Doğan, Sun Kim, Donald C. Comeau, Won Kim, Lana Yeganova, Zhiyong Lu, and W. John Wilbur. Author name disambiguation for PubMed. *Journal of the Association for Information Science and Technology*, 65(4):765–781, 2014. doi:10.1002/asi.23063.

[LKBJ12] Michael Levin, Stefan Krawczyk, Steven Bethard, and Dan Jurafsky. Citation-based bootstrapping for large-scale author disambiguation. *JASIST*, 63(5):1030–1047, 2012. doi:10.1002/asi.22621.

[LKF07] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1), 2007. doi:10.1145/1217299.1217301.

[LR06] Michael Ley and Patrick Reuther. Maintaining an Online Bibliographical Database: The Problem of Data Quality. In *EGC 2006: Extraction et gestion des connaissances, Actes des sixièmes journées Extraction et Gestion des Connaissances*, volume RNTI-E-6 of *Revue des Nouvelles Technologies de l'Information*, pages 5–10. Cépaduès-Éditions, 2006.

[MCM15] Catarina Moreira, Pável Calado, and Bruno Martins. Learning to rank academic experts in the DBLP dataset. *Expert Systems*, 32(4):477–493, 2015. doi:10.1111/exsy.12062.

[ME97] Alvaro E. Monge and Charles Elkan. An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records. In *DMKD 1997 at SIGMOD'97: Workshop on Research Issues on Data Mining and Knowledge Discovery, Tucson, AZ, USA*, 1997.

[MGLP09] Waister Silva Martins, Marcos André Gonçalves, Alberto H. F. Laender, and Gisele L. Pappa. Learning to assess the quality of scientific conferences: a case study in computer science. In *JCDL '09: Proc. of the 2009 Joint Int. Conf. on Digital Libraries, Austin, TX, USA*, pages 193–202. ACM, 2009. doi:10.1145/1555400.1555431.

[Mil11] Steven J. Miller. *Metadata for Digital Collections: A How-to-do-it Manual*. How-to-do-it manuals for libraries. Neal-Schuman Publishers, New York, USA, 2011.

[MM16] Fakhri Momeni and Philipp Mayr. Evaluating Co-authorship Networks in Author Name Disambiguation for Common Names. In *TPDL '16: 20th Int. Conf. on Theory and Practice of Digital Libraries, Hannover, Germany*, volume 9819 of *LNCS*, pages 386–391. Springer, 2016. doi:10.1007/978-3-319-43997-6_31.

[MMR14] Helena Mihaljevic-Brandt, Fabian Müller, and Nicolas Roy. Author Profile Pages in zbMATH - Improving Accuracy through User Interaction. In *CICM '14: Joint Proc. of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM, Coimbra, Portugal*, volume 1186 of *CEUR Workshop Proc.* CEUR-WS.org, 2014.

[MRR17] Mark-Christoph Müller, Florian Reitz, and Nicolas Roy. Data sets for author name disambiguation: an empirical analysis and a new resource. *Scientometrics*, 111(3):1467–1500, 2017. doi:10.1007/s11192-017-2363-5.

[MSP10] Dana McKay, Silvia Sanchez, and Rebecca Parker. What's my name again?: Sociotechnical considerations for author name management in research databases. In *OZCHI '10: Proc. of the 22nd Australasian Computer-Human Interaction Conf., Brisbane, Australia*, pages 240–247. ACM, 2010. doi:10.1145/1952222.1952274.

[New04] Mark E. J. Newman. Who Is the Best Connected Scientist? A Study of Scientific Coauthorship Networks. In *Complex Networks*, volume 650 of *Lecture Notes in Physics*, pages 337–370. Springer, 2004. doi:10.1007/978-3-540-44485-5_16.

[NIS04] NISO. *Understanding metadata*. National Information Standards Organization, 2004. ISBN 1-880124-62-9.

[OEL+06] Byung-Won On, Ergin Elmacioglu, Dongwon Lee, Jaewoo Kang, and Jian Pei. Improving Grouped-Entity Resolution Using Quasi-Cliques. In *ICDM '06: Proc. of the 6th IEEE Int. Conf. on Data Mining, Hong Kong, China*, pages 1008–1015. IEEE Computer Society, 2006. doi:10.1109/ICDM.2006.85.

[OLKM05] Byung-Won On, Dongwon Lee, Jaewoo Kang, and Prasenjit Mitra. Comparative study of name disambiguation problem using a scalable blocking-based framework. In *JCDL '05: Proc. of the Joint Conf. on Digital Libraries, Denver, CO, USA*, pages 344–353. ACM, 2005. doi:10.1145/1065385.1065463.

[PDFV05] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, 2005. doi:10.1038/nature03607.

[Pet05] Luuk Peters. Change detection in XML trees: a survey. In *3rd Twente Student Conf. on IT*, 2005.

[Pop93] Hans Popst, editor. *Regeln für die alphabetische Katalogisierung in wissenschaftlichen Bibliotheken*. Dt. Bibliotheksinst., Berlin, 2. edition, 1993.

[PRZ+09] Denilson Alves Pereira, Berthier A. Ribeiro-Neto, Nivio Ziviani, Alberto H. F. Laender, Marcos André Gonçalves, and Anderson A. Ferreira. Using web information for author name disambiguation. In *JCDL '09: Proc. of the Joint Int. Conf. on Digital Libraries, Austin, TX, USA*, pages 49–58. ACM, 2009. doi:10.1145/1555400.1555409.

[QHC+11] Ya-nan Qian, Yunhua Hu, Jianling Cui, Qinghua Zheng, and Zaiqing Nie. Combining machine learning and human judgment in author disambiguation. In *CIKM '11: Proc. of the 20th Conference on Information and Knowledge Management, Glasgow, United Kingdom*, pages 1241–1246. ACM, 2011. doi:10.1145/2063576.2063756.

[QZS+15] Ya-nan Qian, Qinghua Zheng, Tetsuya Sakai, Junting Ye, and Jun Liu. Dynamic author name disambiguation for growing digital libraries. *Inf. Retr. Journal*, 18(5):379–412, 2015. doi:10.1007/s10791-015-9261-3.

[RCC+04] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *Proc. of the National Academy of Sciences*, 101(9):2658, 2004. doi:10.1073/pnas.0400054101.

[Red96] Thomas C. Redman. *Data Quality for the Information Age*. Artech House, Inc., Norwood, MA, USA, 1st edition, 1996.

[Rei10] Florian Reitz. A framework for an ego-centered and time-aware visualization of relations in arbitrary data repositories. *CoRR*, abs/1009.5183, 2010.

[Rei18] Florian Reitz. Two Test Collections for the Author Name Disambiguation Problem based on DBLP, March 2018. doi:10.5281/zenodo.1201324.

[Reu07] Patrick Reuther. *Namen sind wie Schall und Rauch: Ein semantisch orientierter Ansatz zum Personal Name Matching*. PhD thesis, University of Trier, Germany, 2007.

[RH10a] Florian Reitz and Oliver Hoffmann. An Analysis of the Evolving Coverage of Computer Science Sub-fields in the DBLP Digital Library. In *ECDL '10: Proc. of the 14th European Conf. on Research and Advanced Technology for Digital Libraries, Glasgow, UK*, volume 6273 of *LNCS*, pages 216–227. Springer, 2010. doi:10.1007/978-3-642-15464-5_23.

[RH10b] Florian Reitz and Oliver Hoffmann. Learning from the Past: An Analysis of Person Name Corrections in DBLP Collection and Social Network Properties of Affected Entities. In *ASONAM '10: Int. Conf. on Advances in Social Networks Analysis and Mining, Odense, Denmark*, pages 9–16. IEEE Computer Society, 2010. doi:10.1109/ASONAM.2010.35.

[RH11] Florian Reitz and Oliver Hoffmann. Did They Notice? - A Case-Study on the Community Contribution to Data Quality in DBLP. In *TPDL '11: Proc. of the Int. Conf. on Theory and Practice of Digital Libraries, Berlin, Germany*, volume 6966 of *LNCS*, pages 204–215. Springer, 2011. doi:10.1007/978-3-642-24469-8_22.

[RH13] Florian Reitz and Oliver Hoffmann. Learning from the Past: An Analysis of Person Name Corrections in the DBLP Collection and Social Network Properties of Affected Entities. In *The Influence of Technology on Social Network Analysis and Mining*, volume 6 of *Lecture Notes in Social Networks*, pages 427–453. Springer, 2013. doi:10.1007/978-3-7091-1346-2_19.

[RW06] Patrick Reuther and Bernd Walter. Survey on test collections and techniques for personal name matching. *IJMSO*, 1(2):89–99, 2006. doi:10.1504/IJMSO.2006.011006.

[Sch16] Jan Schulz. Using Monte Carlo simulations to assess the impact of author name disambiguation quality on different bibliometric analyses. *Scientometrics*, 107(3):1283–1298, 2016. doi:10.1007/s11192-016-1892-7.

[SGLF15] Alan Filipe Santana, Marcos André Gonçalves, Alberto H. F. Laender, and Anderson A. Ferreira. On the combination of domain-specific heuristics for author name disambiguation: the nearest cluster method. *Int. J. on Digital Libraries*, 16(3-4):229–246, 2015. doi:10.1007/s00799-015-0158-y.

[SH10] Pnina Shachaf and Noriko Hara. Beyond vandalism: Wikipedia trolls. *J. Information Science*, 36(3):357–370, 2010. doi:10.1177/0165551510365390.

[SHC+07] Yang Song, Jian Huang, Isaac G. Councill, Jia Li, and C. Lee Giles. Efficient topic-based unsupervised name disambiguation. In *JCDL '07: Proc. of the Joint Conf. on Digital Libraries, Vancouver, BC, Canada*, pages 342–351. ACM, 2007. doi:10.1145/1255175.1255243.

[SKCK14] Dongwook Shin, Taehwan Kim, Joongmin Choi, and Jungsun Kim. Author name disambiguation using a graph model with node splitting and merging based on bibliographic information. *Scientometrics*, 100(1):15–50, 2014. doi:10.1007/s11192-014-1289-4.

[SKG+03] Alan F. Smeaton, Gary Keogh, Cathal Gurrin, Kieran McDonald, and Tom Sødring. Analysis of papers from twenty-five years of SIGIR Conf.s: what have we been doing for the last quarter of a century? *SIGIR Forum*, 37(1):49–53, 2003. doi:10.1145/945546.945550.

[SKJC10] Dongwook Shin, Taehwan Kim, Hana Jung, and Joongmin Choi. Automatic Method for Author Name Disambiguation Using Social Networks. In *AINA '10: 24th Int. Conf. on Advanced Information Networking and Applications, Perth, Australia*, pages 1263–1270. IEEE Computer Society, 2010. doi:10.1109/AINA.2010.66.

[SLK16] Jae-Wook Seol, Seok-Hyoung Lee, and Kwang-Young Kim. Author Disambiguation Using Co-Author Network and Supervised Learning Approach in Scholarly Data. *Int. Journal of Software Engineering and Its Applications*, 10(4):73–82, 2016.

[SLM09] Liangcai Shu, Bo Long, and Weiyi Meng. A Latent Topic Model for Complete Entity Resolution. In *ICDE '09: Proc. of the 25th Int. Conf. on Data Engineering, Shanghai, China*, pages 880–891. IEEE Computer Society, 2009. doi:10.1109/ICDE.2009.29.

[SLM10] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3), 2010. doi:10.1145/1670679.1670680.

[SSK+16] Chenchen Sun, Derong Shen, Yue Kou, Tiezheng Nie, and Ge Yu. Topological Features Based Entity Disambiguation. *J. Comput. Sci. Technol.*, 31(5):1053–1068, 2016. doi:10.1007/s11390-016-1679-6.

[TFWZ12] Jie Tang, Alvis Cheuk M. Fong, Bo Wang, and Jing Zhang. A Unified Probabilistic Framework for Name Disambiguation in Digital Library. *IEEE Trans. Knowl. Data Eng.*, 24(6):975–987, 2012. doi:10.1109/TKDE.2011.13.

[TG09] Pucktada Treeratpituk and C. Lee Giles. Disambiguating authors in academic publications using random forests. In *JCDL '09: Proc. of the 2009 Joint Int. Conf. on Digital Libraries, Austin, TX, USA*, pages 39–48. ACM, 2009. doi:10.1145/1555400.1555408.

[TG12] Pucktada Treeratpituk and C. Lee Giles. Name-Ethnicity Classification and Ethnicity-Sensitive Name Matching. In *AAAI '12: Proc. of the 26. Conf. on Artificial Intelligence, Toronto, Canada*. AAAI Press, 2012.

[TKL06] Yee Fan Tan, Min-Yen Kan, and Dongwon Lee. Search engine driven author disambiguation. In *JCDL '06: Proc. of the Joint Conf. on Digital Libraries, Chapel Hill, NC, USA*, pages 314–315. ACM, 2006. doi:10.1145/1141753.1141826.

[TS09] Vetle I. Torvik and Neil R. Smalheiser. Author name disambiguation in MEDLINE. *TKDD*, 3(3), 2009. doi:10.1007/10.1145/1552303.1552304.

[UI01] National Information Standards Organization (U.S.) and American National Standards Institute. *The Dublin Core Metadata Element Set: an American national standard*. National information standards series. NISO Press, 2001.

[VFG+12] Adriano Veloso, Anderson A. Ferreira, Marcos André Gonçalves, Alberto H. F. Laender, and Wagner Meira Jr. Cost-effective on-demand associative author name disambiguation. *Inf. Process. Manage.*, 48(4):680–697, 2012. doi:10.1016/j.ipm.2011.08.005.

[vL89] E.L. Bibliotheksverband der DDR. Kommission für Katalogfragen von Oppen and Bibliographisches Institut Leipzig. *Regeln für die Alphabetische Katalogisierung (RAK)*. Bibliographisches Institut Leipzig, 1989.

[WBH+12] Jian Wang, Kaspars Berzins, Diana Hicks, Julia Melkers, Fang Xiao, and Diogo Pinheiro. A boosted-trees method for name disambiguation. *Scientometrics*, 93(2):391–411, 2012. doi:10.1007/s11192-012-0681-1.

[WD13] Jiang Wu and Xiu-Hao Ding. Author name disambiguation in scientific collaboration and mobility cases. *Scientometrics*, 96(3):683–697, 2013. doi:10.1007/s11192-013-0978-8.

[Win99] William E. Winkler. The State of Record Linkage and Current Research Problems. Technical report, Statistical Research Division, U.S. Census Bureau, 1999.

[WLPH14] Hao Wu, Bo Li, Yijian Pei, and Jun He. Unsupervised author disambiguation using Dempster-Shafer theory. *Scientometrics*, 101(3):1955–1972, 2014. doi:10.1007/s11192-014-1283-x.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998. doi:10.1038/30918.

[WTCY11] Xuezhi Wang, Jie Tang, Hong Cheng, and Philip S. Yu. ADANA: Active Name Disambiguation. In *ICDM '11: 11th Int. Conf. on Data Mining, Vancouver, BC, Canada*, pages 794–803. IEEE Computer Society, 2011. doi:10.1109/ICDM.2011.19.

[YLKG07] Su Yan, Dongwon Lee, Min-Yen Kan, and C. Lee Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *JCDL '07: Proc. of the 2007 Joint Conf. on Digital Libraries, Vancouver, BC, Canada*, pages 185–194. ACM, 2007. doi:10.1145/1255175.1255213.

[YPJ+08] Kai-Hsiang Yang, Hsin-Tsung Peng, Jian-Yi Jiang, Hahn-Ming Lee, and Jan-Ming Ho. Author Name Disambiguation for Citations Using Topic and Web Correlation. In *ECDL '08: Proc. of the 12th European Conf. on Research and Advanced Technology for Digital Libraries, Aarhus, Denmark*, volume 5173 of *LNCS*, pages 185–196. Springer, 2008. doi:10.1007/978-3-540-87599-4_19.

[ZA10] Michael A. Zarro and Robert B. Allen. User-Contributed Descriptive Metadata for Libraries and Cultural Institutions. In *ECDL '10: Proc. of the 14th European Conf. on Research and Advanced Technology for Digital Libraries, Glasgow, UK*, volume 6273 of *LNCS*, pages 46–54. Springer, 2010. doi:10.1007/978-3-642-15464-5_7.

[ZDH16] Baichuan Zhang, Murat Dundar, and Mohammad Al Hasan. Bayesian Non-Exhaustive Classification A Case Study: Online Name Disambiguation using Temporal Record Streams. In *CIKM '16: 25th Int. Conf. on Information and Knowledge Management, Indianapolis, IN, USA*, pages 1341–1350. ACM, 2016. doi:10.1145/2983323.2983714.

[ZQ08] Marcia Lei Zeng and Jian Qin. *Metadata*. Neal-Schuman Publishers, New York, 2008.

[ZYX+14] Jia Zhu, Yi Yang, Qing Xie, Liwei Wang, and Saeed-Ul Hassan. Robust hybrid name disambiguation framework for large databases. *Scientometrics*, 98(3):2255–2274, 2014. doi:10.1007/s11192-013-1151-0.