Data Integration with Dynamic Data Sources

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

ANJA GRUENHEID

M.Sc. in Computer Science, TU Munich
M.Sc. in Computer Science, Georgia Institute of Technology

born on 23.05.1987

citizen of Germany

accepted on the recommendation of

Prof. Dr. Donald Kossmann, examiner
Prof. Dr. Thomas Hofmann, co-examiner
Divesh Srivastava, PhD, co-examiner

2016
Data integration is the task of collecting data from various data sources and combining this data in a meaningful way. Traditionally, it is achieved by first extracting the data, for example, by taking snapshots of the current dataset. Afterwards, the data is transformed through batch algorithms that perform tasks such as entity resolution or data cleaning to format the data according to the user’s preferences. Finally, the integrated data is loaded into a new system. This three-step process is referred to as ETL process and is traditionally applied on static datasets. In this dissertation, we examine how data integration processes need to be modified if data sources are dynamically changing the data they contain. More specifically, modern data integration systems aim to provide integrated data in near real-time where changes to the underlying dataset are propagated with little delay. In this work, we therefore discuss algorithmic changes to existing data integration algorithms that are applied in the transform part of the ETL process, we examine changes to these tasks given different data sources and data interaction requirements, and finally, we propose design principles for such online data integration systems.

This dissertation has two parts, data integration methodology and data integration system design. Data integration methodology discusses modifications to the existing data integration pipeline due to a) continuously changing datasets and b) novel ideas on how to interact with such data. As traditional data integration algorithms are designed for batch processing, we initially focus on how we can adapt existing techniques to incrementally process data, i.e., how to modify only those parts of the integrated dataset that are actively changing. We use as an example the integration task of entity resolution (ER) which is the task of finding those records in a dataset that refer to the same real-world entity. We then show how we can transform widely applied ER algorithms to process continuous data efficiently and without loss of quality. This line of work is then extended for not only automatically processing data but to enable ER tasks that use crowdsourced
data as a data source. In contrast to automatically generated data, crowdsourced data may contain contradictions which need to be handled appropriately to not distort the integrated solution. For that purpose, our work develops graph-based ER strategies that not only help to determine a good ER solution given a set of crowdsourced data but can also be used to establish interactive ER pipelines that minimize the cost of crowdsourcing, i.e., how often humans have to provide data for the ER process. The notion of interactive data access has become more important with the increase in online systems that provide users with insight into their own data. In our work, we address the interactive integration space by looking at a second data integration task called data cleaning. The goal of data cleaning is to automatically determine those parts of the data that are not aligned with the remaining dataset and to mark them for verification. However, finding so-called dirty data is an inherently difficult task which is time-consuming and thus often done as an offline batch process. To address this problem, we introduce a new similarity metric in our work which determines clean data items based on their context in linear time.

In the second part of our work, we focus on conceptual design elements that enable a good online data integration system. As mentioned previously, a lot of research done in recent years has focused on modifying traditional data integration systems to suit novel data characteristics. However, these systems were originally designed with a different set of goals in mind than online data integration systems. Our work shows through the development of our own online event integration the design principles that apply to this novel type of integration system. Specifically, we discuss different aggregation and parallelization strategies that help to process incoming data in near real-time. At the same time, we combine these performance-oriented techniques with incremental integration methods to obtain good quality integration results.

This dissertation contributes to understanding and addressing the problem of online data integration for existing as well as for novel data integration systems that are designed with online use cases in mind. It provides essential ideas for both advancing and transforming existing data integration tasks and describes how new data integration systems can leverage performance-conscious design ideas to integrate continuously changing data.
Zusammenfassung


Die Dissertation enthält zwei Teile: Datenintegrations-Methodik und Systemdesign für Datenintegration. Der erste Teil beschreibt, wie bestehende Algorithmen verändert werden können, um Datenintegration mit sich ständig ändernden Datensätzen zu ermöglichen und diskutiert neue Ideen für die Interaktion mit solchen Daten. Da bestehende Systeme bisher auf statischen Datensätzen arbeiteten, besprechen wir zuerst, wie inkrementelle Datenänderungen verwirklicht werden können. Wir zeigen diese Änderungen am Beispiel von Objektidentifizierung (OI). Dies ist ein Algorithmus, welcher in einem Datensatz nach


Die Dissertation soll zum Verständnis und zur Lösung des Problems der dynamischen Datenintegration in existierenden und neuen anwendungsspezifischen Systemen beitragen. Sie beinhaltet essentielle Ideen zur Weiterentwicklung von vorhandener Datenintegrations-Methodik und beschreibt, wie neue Systeme entwickelt werden können, die dynamische Datensätze effizient bearbeiten.
Acknowledgments

This dissertation would not have been possible without the support of several people. First, I would like to thank my adviser Prof. Donald Kossmann who allowed me to pursue a doctoral degree at ETH and who gave me the freedom to follow my research interests and helped me to become a better researcher. I would also like to thank my co-adviser Divesh Srivastava who was always available to discuss ideas and thoughts and to guide me throughout the last five years. During my years at ETH, I have collaborated with several researchers who inspired me and helped me along the way. In chronological order, I would first like to thank Luna Dong who substantially contributed to the work done on incremental entity resolution described in the first part of this dissertation. I want to also thank Besmira Nushi with whom I worked closely together during the first two years of my doctoral studies and who was always there to give me feedback and to support me. When I first thought about a novel data integration system with online data, one of the main challenges was to find suitable data sources and I would like to thank Theodoros Rekatsinas without whose help this project would have not run as smoothly as it has. Finally, I am grateful for the opportunity to have interned with Microsoft Research and Google NYC during my studies.

Next to the people who actively contributed to the content of this dissertation, there are many people that supported me over the last years. First, I would like to thank my family for being there for me and understanding me. There also have been several people that made my time at ETH a lot of fun. Therefore, I would like to especially thank (again) Besa, Jana, Theo, and Qin for their constant support and Darko, Gerd, Pratamu, Pravin, Simon, and Stefan for a lot of fun ski trips, (after) conference trips and time spent outside of work.
# Contents

## 1 Introduction

1.1 Problem Statement .................................................. 5
   1.1.1 Data Integration Methodology .............................. 6
       1.1.1.1 Extension of Existing Algorithms ................. 6
       1.1.1.2 Novel Algorithms for Dynamic Data Integration .... 8
   1.1.2 Data Integration System Design ............................ 9

1.2 Contributions .......................................................... 9

1.3 Structure of the Dissertation ...................................... 11

## 2 Background

2.1 Automated Data Integration ........................................ 14
   2.1.1 Schema Alignment ............................................ 14
       2.1.1.1 Probabilistic Schema Alignment .................. 16
       2.1.1.2 Scalable Schema Alignment ....................... 16
   2.1.2 Entity Resolution ............................................ 18
       2.1.2.1 End-to-End Entity Resolution .................... 19
       2.1.2.2 Global Objective Functions for Graph Clustering .... 20
       2.1.2.3 Incremental Entity Resolution .................... 22

2.1.3 Data Fusion .......................................................... 24
   2.1.3.1 Data Cleaning .......................................... 25
Contents

2.2 Crowdsourcing ......................................................... 27
  2.2.1 Crowd Quality ............................................. 27
  2.2.2 Crowdsourced Database Operators ....................... 28
  2.2.3 Crowdsourced Entity Resolution ......................... 28

2.3 Alternatives to Online Data Integration Systems ............ 32

3 Data Integration Methodology .................................. 35
  3.1 Entity Resolution ............................................ 36
    3.1.1 Incremental Entity Resolution ......................... 37
      3.1.1.1 Problem Overview ................................ 39
      3.1.1.2 Optimal Incremental Entity Resolution .......... 41
      3.1.1.3 Greedy Incremental Entity Resolution ........... 54
      3.1.1.4 Experimental Evaluation ......................... 60
    3.1.2 Fault-Tolerant Entity Resolution with the Crowd ...... 74
      3.1.2.1 Problem Overview ................................ 76
      3.1.2.2 Data Interpretation Problem ..................... 80
      3.1.2.3 ER Algorithm .................................... 88
      3.1.2.4 Next-Crowdsource Problem ....................... 94
      3.1.2.5 Experimental Evaluation ......................... 99
  3.2 Data Cleaning .................................................... 117
    3.2.1 Problem Overview ..................................... 119
    3.2.2 Outlier Identification .................................. 122
    3.2.3 Outlier Detection ...................................... 129
    3.2.4 Token Corrections ...................................... 133
    3.2.5 Experimental Evaluation ............................... 136

4 Data Integration System Design .................................. 149
  4.1 Problem Overview ............................................. 151
## Contents

4.2 Terminology ........................................... 153  
4.3 Integration Quality of StoryPivot .................... 157  
4.4 System Performance of StoryPivot ................. 162  
4.5 Experimental Evaluation ............................ 166  
4.6 Discussion ............................................ 179  

5 Conclusion ............................................. 181  
  5.1 Data Integration Methodology ....................... 181  
  5.2 Data Integration System Design ................. 183
Data integration is traditionally seen as the task of combining various types of data across multiple data sources. This is commonly done by integrating snapshot data in so-called extract, transform, and load (ETL) processes and has been an integral part of enterprise data processing over the last decades. In recent years, research has shifted its focus from this traditional data integration setup, i.e., static datasets and batch processing, to continuous, incremental (and thus dynamic) data integration techniques [DS15]. That is, data integration systems are now perceived to be online systems that propagate updates to the original datasets in near real-time into the integrated solution which shows a complete picture of the data in the system.

To address these novel requirements for data integration systems, research has focused on two different areas. First, given the fundamentally different design aspects of a dynamic data integration system, completely new algorithms and systems are being developed. Challenges for these encompass performance-oriented aspects such as scalability and near real-time integration mechanisms while at the same time maintaining high quality integration results. Another aspect that has become increasingly important is the effective use of varying types of data either as direct input to the system or influencing the integration procedure. Specifically, integration systems do not only handle computer-generated, mostly structured data but novel algorithms also leverage data derived from data sources such as the crowd, i.e., human workers that are employed to solve tasks on demand. Second, next to developing novel techniques directly suited to the dynamic integration context, re-
searchers explore how traditional data integration algorithms can be modified to suit this dynamic environment. Both research directions are imperative as one advances systems to be able to manage and process big data efficiently while the other enables existing systems to be adapted without interrupting ongoing processes. We discuss and contribute to both of these research areas through the work described in this dissertation.

Take as an example of the transition of static to dynamic data integration research one of the most commonly executed data integration tasks, entity resolution (ER). Entity resolution is the problem of finding those records in the dataset that point to the same real-world entity. It is commonly implemented through batch clustering algorithms that leverage pair-wise similarities between the records to determine clusters which represent the (supposed) real-world entities. An example for a typical entity resolution scenario is shown in Figure 1.1 (a) where records $r_1 \ldots r_{10}$ resemble entries in a business listing and Figure 1.1 (b) visualizes the corresponding ER solution. Here, a record contains the name, address, and telephone number of a business. An edge between records then signifies how similar the content of two records is. If records are in the same cluster, we say that they belong to the same real-world entity. If the velocity of such a dataset is high, meaning that records are modified, added, or removed at a fast pace, batch clustering the data becomes...
Table (a) shows the business listings before the update and table (b) after the update.

For example, if update $\Delta D_1$ in Figure 1.2 (a) is applied to the original dataset with a batch algorithm, it would recompute the clustering with now eleven records. However, we observe that $r_{11}$ is not closely correlated with any of the other records and will not change the clustering except that it adds a singleton cluster $\{r_{11}\}$ to the clustering in Figure 1.1 (b). Recomputing the whole clustering is therefore unnecessarily expensive.

To avoid these inefficiencies, we discuss incremental techniques that only focus on those parts of the ER solution that have been modified by the changes to the dataset. These techniques do not modify the original algorithm but only enable them to handle fast-paced data changes. Using these mechanisms, we are able to compute the final ER solution as shown in Figure 1.2 (b) in a fraction of the time needed for batch processing the same data increments.

Incremental techniques for data integration are mainly applied for those areas of automated data integration where procedures and algorithms are well established. However, new types of data also pose new challenges to data integration as to how this data can be processed. One example of a novel type of data is crowdsourced data. In the context of entity resolution, crowdsourcing has become a widely applied alternative to traditional
automatic ER algorithms. The goal of crowdsourcing is to enhance automatic algorithms for those kind of tasks that humans are currently better at solving than computer systems, for example picture classification, or where certainty in the ER results is imperative, for example for Facebook’s knowledge graph [Fac16]. Specifically, research on crowdsourced ER often seen as an optimization problem where result quality is maximized while the number of times the crowd is accessed is simultaneously minimized. These optimizations are not trivial as information that is obtained through the crowd actively or passively, for example by parsing the web, is not always correct. Instead, human workers may make (unintentional) mistakes, which result in data that contains errors and contradictions. We refer to this data as noisy in the following. Our work extends research on this novel type of data integration to include the notion of noisy data.

Noisy data is not only a challenge for entity resolution but also for data cleaning, another task that is also part of the data integration process. It is a crucial problem and has substantial influence on other data integration tasks because the better the quality of the data, the higher the accuracy of processing techniques such as ER algorithms. Thus, if we are able to improve the data ingested by the system, we are able to provide better quality services to any user or automatic task of/inside the data integration system. A first step towards repairing erroneous data is to identify potentially wrong data items. Take as an example Figure 1.3 which shows a web table that a user is constructing at this moment. We observe two different types of errors that a user can make here: Syntactic and semantic errors. Syntactic errors are often misspelled words, the example in Figure 1.3 is the misspelled word ATU which should have been AUT (the country code of Austria). Semantic errors are those kind of errors that often occur in large-scale data integration where data from various data sources is integrated and the formatting or meaning is different in each source. In Figure 1.3, we observe that word Switzerland contextually
is not wrong as it is the correct country for the row. However, all other entries in this column have a different formatting which makes it an outlier. Outlier detection is not a novel problem and has been addressed in a variety of research areas, for example in machine learning. However, these approaches are focused on static data processing. In contrast, we examine outlier detection for online integration systems, i.e., when users want to interact with the system in real-time.

All of these methodological improvements explore and address specific facets of the data integration process. However, they target distinct problems, here either entity resolution or data cleaning, and are not generally applicable for all types of data integration tasks. Therefore, we expanded our work from research on algorithms for dynamic data sources to research on general systems for data integration with high data velocity. Leveraging the techniques that we previously developed for these data integration tasks, we determine the design challenges for dynamic data integration systems that are designed from scratch. We mainly focus on the question whether there exists a fundamental difference for this type of system compared to systems that do data integration with static datasets and pay attention to the trade-off between system performance and result quality. The results that we present in this part of our work are generally applicable for a new generation of data integration systems that by design enable the management of large amounts of fast-changing data without losing integration quality.

We strongly believe that adjusting algorithms to the dynamic setting as described previously is a necessary step that bridges the gap until fully functional data integration systems for dynamically changing data have been developed. Thus, these two areas of research go hand in hand to transform traditional data integration mechanisms and systems into data integration technology that is able to manage dynamic data sources.

1.1 Problem Statement

The work described in this dissertation can be separated using two different means of comparison. First, we differentiate between extending existing technologies and inventing novel mechanisms for dynamic data integration. Second, our work can be split into algorithmic and system-based contributions.

As explained previously, our work addresses the transition of static data integration mechanisms to algorithms and systems that are able to handle dynamic datasets. In the first
part of this dissertation, we will therefore discuss data integration methodology, i.e., how
batch algorithms can be extended to incremental algorithms and how novel types of data
can be effectively addressed with novel data integration techniques. Data integration
methodology thus represents the algorithmic part of the second dimension and addresses
both areas of the first dimension. The second part of our work focuses on the system as-
pect of dynamic data integration and discusses a novel approach to online data integration.
Specifically, we describe through the example of our own event processing system which
design features are crucial to modern data integration systems. The challenges associated
with each of these parts are shortly outlined next.

1.1.1 Data Integration Methodology

Our contributions to data integration methodology focus on how we can extend exist-
ing algorithms to make them more efficient and to make them suitable for varying data
types. We describe the different problem statements that this dissertation addresses in
this context in the following.

1.1.1.1 Extension of Existing Algorithms

Data integration is not a novel research area, indeed, it has been subject to various research
over more than three decades. Thus, a lot of highly specialized techniques for different
tasks in data integration, entity resolution and data cleaning amongst others, are already
well-established in existing ETL processes or integration systems. However, most of these
algorithms have been designed as batch processes. The reason is that data integration was
traditionally executed on top of snapshot data. Between snapshots, data from different
data sources was accumulated and then combined at a specific point in time to provide
a consistent integrated view of the data. The obvious drawback of this approach is that
whenever a data item in the underlying data changes, its modification is not immediately
propagated to the integrated solution but only whenever the next snapshot is taken. Thus,
it is possible that the integrated data becomes stale or even that updates are never shown
in the integrated solution because a data item is changed more than once in the interval
between snapshots.

Dynamically updating the integrated data whenever a data item is changed is the obvious
solution to this problem. But how is that possible if the currently utilized algorithms are
executed as batch processes? This is exactly the problem that we address in this part of our work. More formally, we define the problem of adapting a batch algorithm to suit a dynamic integration environment as follows.

**Problem 1.1.1** (Adaptation of Static Batch Algorithms). Let $A$ be a data integration algorithm that batch processes data. If $A'$ is a modification to the original algorithm $A$, we expect that the execution time of $A'$ is significantly better than $A$, i.e., $\text{perf}(A') \ll \text{perf}(A)$. Additionally we expect that the result quality of $A'$ is equivalent to the output quality of $A$, meaning that $\text{qual}(A') = \text{qual}(A)$ holds.

To address this problem, we look at mechanisms for adapting static batch algorithms of one specific group of data integration algorithms, namely entity resolution algorithms. Our work has been published in [GDS14] and examines the transition of two well-established ER algorithms, correlation clustering [BBC04] and DB-Index [DB79], to dynamic, incremental algorithms. Specifically, we propose modifications to the algorithms that limit their execution to only those parts of the ER solution that have been (indirectly) changed by the updates to the underlying dataset. These mechanisms reduce their execution time significantly. At the same time, we formally guarantee that the quality of the final ER solution is equivalent to the quality of the original batch algorithm.

Automatic adjustment of the ER solution is not the only means to incrementally modify the solution. An alternative is to apply crowdsourcing to enhance the existing solution or to construct a new ER solution if the relationship of records was previously unknown. Crowdsourcing is the concept of asking human workers to resolve a task that commonly a) a computer is not as adequate at executing as the human worker or b) is crucial to some business process where result quality is imperative. It has been widely applied to improve data quality in recent years using often on-demand crowdsourcing platforms such Amazon Mechanical Turk [Ama16] or CrowdFlower [Cro16]. However, crowdsourcing is expensive as the workers who do the tasks need to be paid. Thus, data integration tasks that leverage crowdsourcing have to pay attention to use this mechanism as effectively as possible. In that context, we formally define the problem of crowdsourced ER as follows.

**Problem 1.1.2** (Crowdsourced Entity Resolution). Given a set of records $R$, crowdsourced entity resolution resolves $R$ into a clustering $C$ where each cluster represents a real-world entity. To do so, the ER algorithm may request data from the crowd of the form ‘Does $r_i$ belong to the same entity as $r_j$?’ where $r_i$ and $r_j$ are records in $R$. The goal then is to maximize the ER solution quality while minimizing the number of accesses to the crowd.
Chapter 1: Introduction

This optimization problem has been discussed in recent work because it has numerous applications in real-world frameworks such as knowledge graphs. However, previous approaches have focused on solving this problem under the assumption that crowd workers answer correctly. Studies have shown that this assumption does not hold on general-purpose crowdsourcing platforms [IPW10]. In our work, we therefore examine the problem of handling these potentially incorrect interactive data sources. We propose a graph traversal algorithm for evaluating whether two records point to the same entity, extending the idea that transitivity can be leveraged to reduce the search space found in previous work. Furthermore, our approach can be easily combined with the incremental entity resolution techniques introduced before to perform efficient graph updates. Our research has been made publicly available in [GNK+15] and [GKRW12] as technical reports.

1.1.1.2 Novel Algorithms for Dynamic Data Integration

One of the areas of data integration that is influenced significantly by dynamic data is data cleaning. As shown above, new types of data - especially crowdsourced data - introduce noise into datasets which makes processing this data with analytical algorithms more difficult. As a result, data cleaning has become a necessity when handling human-generated data and more generally any type of data that may contain errors. To address data cleaning on dynamic data, we examine online outlier detection mechanisms in our work. An outlier here is defined as a (set of) value(s) that do not fit the dataset. In the introductory example for this kind of problem, Figure 1.3, we have shown two types of erroneous values that can commonly be identified in the data cleaning process. More formally, we describe the task of outlier detection as follows.

**Problem 1.1.3 (Outlier Detection).** Given a set of tokens \( R \), i.e., words or a combination thereof, we want to find those tokens \( r_i \in R \) that do not fit the context of the other tokens \( R \setminus r_i \). In other words, we want to find a clustering of (groups of) tokens given an input set \( R \) where each cluster is consistent.

To find the context of a token, we leverage information that can be found in structured data such as web tables or business spreadsheets within enterprises. We observe that this kind of data often contains semantically consistent content as has been shown in previous work for web tables [CHW+08]. This knowledge can then be used to build a framework that allows us to judge the semantic consistency of the input tokens and identify outliers in near real-time.
1.1.2 Data Integration System Design

The adaptation of existing algorithms as well as the application of novel algorithms for dynamic data sources enhances the functionality of current data integration systems. However, all of the techniques presented in the previous sections are only applicable for a specific part of the data integration process. As a result, one of the interesting questions to look at is how these techniques and their management of dynamic data sources can be generalized. That is, we aim to design generally applicable system elements that are suited for online data integration over time and can be leveraged for several data integration tasks.

**Problem 1.1.4** (Design of a Dynamic Data Integration System). *Under the assumption of fast-changing data sources, designing a data integration system focuses on two core aspects: quality and performance. Previously, the application of batch algorithms guaranteed high quality but (comparatively) low performance. Given new and adapted integration algorithms, the question then is whether and how a redesigned integration framework can enhance their effectiveness for continuous integration processes.*

We address this problem through the implementation of our own dynamic event integration system, StoryPivot. It is an event processing system that integrates news data from various data sources such as the New York Times, The Guardian, BBC etc. in near real-time. To evaluate the trade-off between quality and performance for such a system, we look at traditional performance enhancing techniques such as parallelization and aggregation and evaluate their impact on the integration quality. We then extend the evaluation to determine whether StoryPivot is by design scalable and could thus be expanded into a distributed data integration system. A description of the key design ideas of StoryPivot can be found in [GKRS15].

**1.2 Contributions**

As described in the previous section, we differentiate the problems that this dissertation addresses in two dimensions. Specifically, the data integration methodology part of this thesis addresses algorithmic challenges and describes the extension of existing techniques in entity resolution as well as introducing novel approaches in the area of data cleaning. The second part of our work, data integration system design, focuses on the other hand on the system dimension, introducing novel design concepts that are generally applicable
Chapter 1: Introduction

for a variety of online data integration systems. An overview of these two dimension and
the categorization of each of the building blocks in this dissertation is shown in Figure 1.4.
In total, this categorization enables four different combinations of the two dimensions: 1) algorithmic vs. system and 2) extension of existing work vs. novel approaches. For each of these combinations, we show the (published) papers that this dissertation incorporates and which are as follows:

I  A. Gruenheid, X. L. Dong, D. Srivastava
   Incremental Record Linkage
   published in PVLDB 7(9), p. 697-708 (2014)

II(a) A. Gruenheid, B. Nushi, T. Kraska, W. Gatterbauer, D. Kossmann
    Fault-Tolerant Entity Resolution with the Crowd
    made publicly available on arXiv in 2015

II(b) A. Gruenheid, D. Kossmann, S. Ramesh, F. Widmer
    Crowdsourcing Entity Resolution: When is A=B?
    made publicly available as ETH Technical Report in 2012

III  A. Gruenheid, Y. He, K. Ganjam
     Outlier Detection through Group Similarity
     currently in submission

IV(a) A. Gruenheid, D. Kossmann, D. Srivastava
     Online Event Integration with StoryPivot
     currently in submission
1.3 Structure of the Dissertation

The remainder of this dissertation is structured as follows.

Chapter 2 - Background. In this chapter, we will give an overview of the historical development of data integration and its transition from static to dynamic management of data. We will further detail current research on the two data integration tasks that this thesis focuses on, entity resolution and data cleaning.

Chapter 3 - Data Integration Methodology. This chapter discusses our contributions in adapting and introducing novel algorithms for dynamic data integration in both entity resolution and data cleaning. It addresses Problems Problem 1.1.1 – Problem 1.1.3 that were previously described.

Chapter 4 - Data Integration System Design. Corresponding to Section 1.1.2, this chapter describes the design features of modern data integration systems that are able to handle fast-paced data changes. It extensively examines the performance characteristics of such a system and evaluates how to efficiently scale such a system.

Chapter 5 - Conclusion and Future Work. In the last chapter of this dissertation, we summarize our contributions and point out future research directions that are a direct result of our efforts.
In this part of the dissertation, we discuss related work in data integration, crowdsourcing, and we explore alternative systems that could be deployed alternatively to data integration systems. We first discuss research on static automated data integration which is a business process that many large-scale businesses have used over the last decades to integrate data stored in various data sources. This process usually has three phases, *schema alignment*, *entity resolution*, and *data fusion*. We outline each of these phases and important research directions in detail in Section 2.1. Furthermore, we highlight research achievements in entity resolution and data cleaning which are research areas closely related to our contributions on incremental entity resolution and outlier detection. Afterwards, we discuss recent advances the area of crowdsourcing, focusing on crowdsourced entity resolution to position our contributions in relation to existing work in this area of research. Finally, data integration systems are not the only systems that can solve the problem of combining data online. For example, streaming systems are an alternative type of system for processing dynamic datasets that have been increasingly deployed in recent years. We thus discuss alternatives to online data integration systems in Section 2.3.


2.1 Automated Data Integration

Traditional data integration addresses three different challenges, semantic ambiguity, instance representation ambiguity, and data inconsistency [DS15]. Semantic ambiguity is the problem of deriving data from multiple data sources and aligning them to conform to one consistent schema. This task is also referred to as schema alignment. After the input data has been standardized, we can perform automated techniques such as entity resolution to address instance representation ambiguity. Finally, once we have identified multiple instances of the same real-world objects, we can unify their representation to address inconsistencies and ambiguities in the input data. We refer to this step as data fusion in the following.

Example 2.1.1. Consider input sources $s_1$, $s_2$, and $s_n$ shown in Figure 2.1. They contain information on capitals of three countries, Austria, Germany, and Switzerland. However, the column naming scheme is dependent on the data source, formatting conventions may differ, and inconsistencies may occur within the same data source and across data sources. In a typical data integration process, we first collect the data from these data sources and align their schemas. Here, we take the schema of $s_1$ and add one further column that contains the country’s abbreviations. Afterwards, we use the aligned schema to identify those tuples that refer to the same object or concept. This is often done through identifying those tuples that are similar and grouping them accordingly. This step results in three different entity groups in our example. In the last phase, we use data fusion techniques to propose a consistent representation of them. This means resolving the inconsistency for the country abbreviation of Austria, the city name of Berlin and Bern, and combining the remaining tuples.

In the following, we provide an overview of current research for each of these data integration tasks and highlight which techniques and problems this dissertation addresses.

2.1.1 Schema Alignment

Schema alignment is essential to data integration systems especially if data integration processes are deployed in business environments and data is derived from multiple data sources. As business systems often evolve over time or get acquired, they commonly do not rely on the same schema when storing their data. As a result, if the user wants to
see an integrated view of her data, she first needs to align the content of the different data sources. Companies such as IBM therefore started developing interactive schema alignment systems two decades ago [HMH01] to enable easy schema alignment for business users. Such systems generally follow a three-step structure. First, a mediated schema is generated, i.e., a target schema to which the data in the source schemas is then mapped in the second step, attribute matching. Mediated schema generation is often a manual process and dependent on the actual data. For example, the ‘Clio’ system by IBM allows schema mediation through an interactive drag and drop interface. Attribute matching can then be done as either a supervised process by experts, i.e., in cases where the system user is already aware of domain-specific matchings, or as an unsupervised process. An overview of different automatic attribute matching techniques is given in [RB01]. Finally, schema mappings are generated that determine how the original data can be queried by the user through the integrated view [Hal01].

With the move from static to dynamic data integration systems, requirements for schema alignment algorithms change. In fact, they now need to address not only the challenge of
variety, i.e., how to combine data from various data sources in a meaningful way, but they also need to consider volume and velocity, i.e., the amount of data processed and how often it changes. We therefore give an overview of probabilistic and scalable schema alignment next, which are two important research areas in the context of schema alignment with dynamic data.

2.1.1.1 Probabilistic Schema Alignment

Probabilistic approaches for schema alignment have been proposed for a) generating the mediated schema and b) for generating the schema mapping. The idea behind these approaches is that the data which needs to be aligned may change over time. Thus, the association between data and the integrated view can be seen as a probabilistic relationship both for generating the view and querying the original data sources based on the integrated view. To generate a probabilistic mediated schema, [DSDH08] proposes to maintain several mediated schemas, each associated with a probability that the schema is accurate. Correctness of these schemas can be confirmed for example through user feedback loops which then refine the corresponding probabilities. Probabilistic schema mapping on the other hand was first introduced in [DHY07]. Here, the authors define the semantics of probabilistic schema mapping and examine the impact of uncertain data on the complexity of querying aligned schemas.

Probabilistic schema alignment is most commonly used to address the velocity aspect of alignment. However, the computation of these approaches is often time-consuming given that there are multiple alternative solutions (matchings or mappings) to evaluate. Thus, alternative schema alignment techniques have emerged that focus on the volume problem, i.e., how data integration systems can handle large amounts of data sources in an efficient way. We discuss these approaches in detail next.

2.1.1.2 Scalable Schema Alignment

With an increase in the volume of available data, structured access to this data has become crucial a) to make it easier to understand and b) to navigate and filter less relevant data. One of the most interesting use cases for such schema alignment techniques is the web because it contains a high volume of structured and unstructured data. Searching for specific data values is thus often a challenge which can be addressed by limiting the available data in a meaningful way. Therefore, many techniques have been developed in
recent years that extract data for example from web tables. This data source has been chosen for a variety of applications because web tables are well-structured and not as prone to extraction mistakes as textual data. It is also one of the reference data sources that we examine in our work on data cleaning in Section 3.2.

Once the data is extracted from these web tables, it is often stored in a mediated schema. In a large number of schema integration systems, the mediated schema is in the same format as to what is called in traditional database management systems a view of the data. Recently, alternative representation schemes have been proposed, namely graph-based knowledge representations or knowledge bases in short. These knowledge bases are used for online query answering similar to the task of schema mapping that we described previously. To highlight the similarities and differences between traditional schema alignment with static data sources and online data sources, we next describe the process of web data extraction, storage, and querying in detail.

**Web Data Extraction**

Structured data cannot only be found in business internal databases but it can be extracted from any type of table. As enterprise data is often not freely available, research in recent years has focused on extracting data from the web. One example of a good data source for structured data are web tables where the header/row structure as well as a table’s metadata can be leveraged for schema alignment. This methodology was first proposed in [CHW+08] and was then extended to determine schema matching with graphical models [LSC10] and semantic annotations to capture value variation over time [ZC13]. Going one step further, techniques that extend the variety of the aligned data sources automatically have been proposed in [DSFG+12]. Here, the authors discuss means to find related web tables given a specific source table. This enables this type of data integration system to extract more targeted data and to expand the system’s integrated view accordingly.

**Knowledge Bases**

Knowledge bases are graph-based storage systems for data where a node is some data item and an edge specifies the relationship of two data items. Often, these are expressed as triples, for example <B. Obama, married-to, M. Obama> where ‘B. Obama’ and ‘M. Obama’ are two distinct data items. Knowledge bases have been developed as open source projects, for example Yago [SKW07, HSB+11] and Freebase [BEP+08], or as commercial
knowledge bases such as used in Google [Goo16] and Facebook [Fac16]. Often, they are not just aligned schemas but they leverage a number of additional integration techniques that we discuss in the following such as data cleaning and fusion to improve the quality of their data items. For example in [DGH+14], the authors propose to combine extracted web information with existing information in the knowledge base using something that the authors refer to as knowledge fusion.

**Online Query Answering**

There are two types of information that users query for and which can be found in web tables. The first type is categorical data. For example, if a user searches for the amount of money spent on buying chocolate in Switzerland, the answer can be found in a web table on the cocoa consumption of different countries. Using this kind of data has been explored in [CHW+08] and [PS12] which leverage the table’s metadata to determine which tables are good candidate matches for the query. Second, web tables provide information on the correlation of data. For example in that cocoa consumption table, one column contains the country names, a second the revenue information and so on. Correlation within a column is especially useful if leveraged for enumeration tasks. For example, if the user wants to enumerate all kinds of cocoa there are, she can start with ‘Forastero’ and ‘Criollo’ and the query engine would respond with ‘Trinitario’. For keyword search queries and set expansion, this technique has been first introduced by [TD08]. In our work described in Section 3.2, we discuss the same kind of correlation but for a different use case: Determining the consistency of data with respect to previously observed web tables.

**2.1.2 Entity Resolution**

Entity resolution can be achieved using two different methodologies. The first one is to identify pair-wise similarities and then using local decision-making to determine entities. This was the initial approach to entity resolution which emerged in 1960’s with work done by Fellegi and Sunter [FS69] amongst others. They developed models to determine whether two records match, maybe match, or do not match based on their pair-wise similarity. The probability of two records referring to the same object can be computed using classifiers as applied in the work above or using string similarity computation such as examined by Elmargind et al in [EIV07]. Examples for algorithms that can then be applied to determine a final entity resolution are then partitioning or transitive closure [HCML09].
2.1: Automated Data Integration

However, there exists an inherent problem with this kind of technique: Once a local decision is wrong, the error will propagate through the result set. To be more robust to erroneous local decisions, the second approach to entity resolution is to apply *global* objective functions instead. Global entity resolution has been extensively studied in the literature, a survey is given in [GM12]. Example global objective functions that have been applied for ER problems are cut [HCML09] or correlation clustering [BBC04]. It is furthermore a topic that is addressed in two parts of this dissertation, Section 3.1.1 and Section 3.1.2. In these, we discuss a) incremental entity resolution which allows us to iteratively compute ER solutions and b) crowdsourced entity resolution where our goal is to minimize the number of crowd accesses while maintaining high quality ER results. To understand how these global objective functions can be leveraged for ER, we next describe an end-to-end ER execution process.

2.1.2.1 End-to-End Entity Resolution

Global entity resolution commonly proceeds in three steps. First, blocking is applied. That means that the records are split into multiple, potentially overlapping blocks where each block contains similar records. That is, a blocking function is applied assigns those records to the same block for which it thinks that they might point to the same real-world entity. This step is especially important if the input data set is large, i.e., if we want to scale our ER execution. In the second step, pair-wise similarity computation is applied to construct the similarity graph \( G = (V, E) \) which documents the probability that two records (represented through nodes in \( G \)) are similar. Finally, a graph clustering is constructed such that a cluster represents a real-world entity. The second and third steps of the process guarantee consistent entities with respect to the chosen objective function. Each of these three steps is discussed in detail next.

**Blocking.** Commonly, blocking is achieved by building an index for the records where each index entry is a block. Thus, each of the blocks can be processed in parallel during the ER execution, improving the scalability of ER algorithms [BD83, BCC03] by reducing the number of pair-wise comparisons within a block. Blocks can be identified for example through words, tokens, k-grams, or domain-specific rules. However, blocking also has a drawback. That is, blocking functions are often domain dependent and if a function is chosen wrongly, the ER algorithm may never evaluate a pair because it is separated in different blocks. To address this problem, techniques that use multiple blocking functions
in parallel or create overlapping blocks emerged [HS98, MNU00]. In our work, we discuss blocking for online linkage problems in further detail when designing our own online data integration system (Chapter 4). It can also be applied in the context of incremental entity resolution. However, as its application does not influence the theoretical results shown in Section 3.1.1, we do not go into any further detail on blocking in that line of work.

**Similarity Computation.** Pair-wise similarity computation can be achieved using either classifier-based techniques or syntactic similarity computation as described previously. Using the probability associated with each pair of records, we then construct the similarity graph $G$ where a record is a node in $G$ and an edge the pair-wise similarity between records. This similarity graph is then the input for the graph clustering phase. We further expand similarity computation from a static set of records to encompass the notion of data increments, i.e., modifications, insertions, or deletions of records in $G$, to enable incremental entity resolution. We refer to $\Delta G$ as the set of records that describe how $G$ changes within an increment. Going back to our initial example in Chapter 1, we see data increments $\Delta D$ depicted in Figure 1.2. In this example, $\Delta D_1$ adds a record to the data set. The corresponding $\Delta G_1$ would thus contain a new node ($r_{11}$) but no new edges as according to the applied similarity function, no other record is sufficiently similar to $r_{11}$ to create such an edge.

**Graph Clustering.** As the last step of the ER process, a global objective function is applied. In static entity resolution, the objective function applied on the similarity graph $G$ while in incremental entity resolution, it is applied on $G + \Delta G$ which is the updated graph after the data increment. As modifying these algorithms for the purpose of incremental entity resolution is one of the core contributions of this dissertation, we will next explain graph clustering using two different objective functions for static similarity graphs.

### 2.1.2.2 Global Objective Functions for Graph Clustering

Graph clustering mechanisms that are used for entity resolution usually leverage an *objective function* and choose the clustering that optimizes the solution according to this function. Two examples for such function-based graph clustering techniques are *correlation clustering* [BBC04] and *DB-index clustering* [DB79]. They penalize high correlation across clusters and low correlation within clusters. This generates dense clusters, i.e., cohesive entities. We choose these two graph clustering algorithms for two reasons. First, they can be applied without knowing the number of entities a priori like for example K-means
clustering [Llo82]. Second, they represent two different categories for ER algorithms as shown in [HCML09]: correlation clustering uses adjacency measures while DB-index uses distance measures in the respective objective function.

### Correlation Clustering

Correlation clustering partitions the nodes in $G$ such that the final solution either maximizes agreements or minimizes disagreements of the edge labels. That is, if an edge $e_{ij}$ is the edge between two nodes $v_i, v_j \in G$, where each node represents a corresponding record $r_i$ and $r_j$, and has value $e_{ij} \in [0, 1]$, correlation clustering can a) decide to maximize the sum of all $e_{ij}$ within a cluster and the inverse of the cutting edge $e_{ij}$ across clusters or b) minimize the inverse of the sum $e_{ij}$ within a cluster and the cut $e_{ij}$ across clusters. Both approaches are equivalent if the goal is to find the optimal ER solution. For the purpose of our work, we focus on minimizing disagreements represented through the penalty that the objective function incurs. Let $L_G$ be the clustering of $G$, and $C$ a cluster in $L_G$. We then compute the penalty function of correlation clustering (CC) as follows.

$$CC(L_G) = \sum_{C \in L_G, v_i, v_j \in C} (1 - e_{ij}) + \sum_{C, C' \in L_G, C \neq C', v_i \in C, v_j \in C'} e_{ij}. \quad (2.1)$$

A special case of correlation clustering is clustering with binary similarities, i.e., two records are either similar and thus $e_{ij} = 1$ or dissimilar meaning $e_{ij} = 0$. For this specific case, correlation clustering has been shown to be NP-complete. The cautious correlation clustering algorithm presented in the original correlation clustering paper [BBC04] can obtain a $9\left(\frac{1}{\delta^2} + 1\right)$-approximation where the algorithm has a complexity of $O(|V|^2)$. The parameter $\delta$ is specific to this algorithm. Using the cautious correlation clustering algorithm, it can also be proven that rounding the edge labels to 0 resp. 1 leads to a $\left(\frac{14}{\delta^2} + 10\right)$-approximation.

### DB-Index Clustering

The original Davies-Bouldin index (DB-index) clustering algorithm [DB79] was defined for Euclidean space but has since then been adapted by [GDSZ10] as follows. It computes the *intra-cluster distance* as the complement of the average similarity between records
Chapter 2: Background

within that cluster while the *inter-cluster distance* is defined as the complement of the average similarity between records across clusters. Specifically, the intra-cluster distance $D$ of a cluster $C$ is computed as $D(C) = 1 - \text{Avg}_{v_i, v_j \in C} e_{ij}$ while the inter-cluster distance $D(C, C')$ is computed as $1 - \text{Avg}_{v_i \in C, v_j \in C'} e_{ij}$. Furthermore, the DB-index uses a *separation measure* which is computed as $M(C, C') = \frac{D(C) + D(C') + \alpha}{D(C) + D(C') + \beta}$. Here, $\alpha$ and $\beta$ are chosen such that the denominator or numerator would affect the result even if the other is 0. For example, consider a graph with two nodes that have an edge similarity of 1. If the nodes are assigned to singleton clusters, i.e., each node is given its own cluster, the separation measure is $\frac{\alpha}{\beta}$. Such a clustering should be penalized highly, thus $\alpha$ should be much larger than $\beta$. For a cluster, the separation measure is defined as $M(C) = \max_{C' \neq C} M(C, C')$. Finally, the DB-index takes all of these measures and combines them as follows.

$$ DB(L_G) = \text{Avg}_{C \in L_G} M(C). \quad (2.2) $$

In [GDSZ10], the authors show that DB-index clustering is intractable and thus approximate it with a hill climbing algorithm that has complexity $O(l|V|^4)$ where $l$ represents the number of iterations the algorithm executes.

### 2.1.2.3 Incremental Entity Resolution

We now explore the line of work that is most closely related to what we refer to as incremental entity resolution. The term incremental entity resolution describes modifying an ER solution not as a batch process, instead limiting the changes to a restricted part of the ER solution. To the best of our knowledge, prior to our work, it had been studied only in [BGMM+09] and [WGM10, WGM14]. The former focuses on data development, it only allows data increments to add to the dataset. However, often records can also be modified or deleted which is not captured in this line of work. The second line of research has focused on evolving matching rules, i.e., how changes of the ER framework can be propagated. Specifically, [WGM14] defines its notion of an increment as follows:

**Definition 2.1.1 (General incremental).** We define $F$ as a batch linkage algorithm whose input is the clustering of records. Let $S(G)$ be the set of singleton clusters for each node in graph $G$. We say $F$ is general incremental if for every subgraph $G' \subseteq G$, we have $F(S(G \setminus G') \cup F(S(G'))) = F(S(G))$. 

22
This definition of a general increment is cluster-based rather than node-based. Specifically, it does not hold if the separation of \( G \) and \( G' \) splits clusters into singleton clusters. However, both of the clustering techniques introduced previously, correlation clustering and DB-Index clustering are node-based clustering techniques. As a result, the definition of generally incrementing their clustering does not hold. Our work discussed in Section 3.1.1 differs from this line of work as follows. First, the definition of an increment is based on the objective function \( O \) instead of the clustering algorithm \( F \). Second, we show that using some general observations on the behavior of node-based clustering functions, locally optimal approximation functions can reasonably mimic the behavior of optimal global objective functions.

Stepping away from a graph-based but time-independent data representation, incremental data changes have also been discussed for evolving data. Evolution is a concept that is most commonly studied for entities such as people, organizations, etc. and it describes the changes to these entities over time. As an example take temporal clustering which is a technique that has been recently employed for record linkage [CDN14, LDMS11]. For example, a professor may take on an appointment at a different university. As she is still the same person, a record linkage system should be able to correlate multiple records that contain this kind of varying but evolving information. Furthermore, this problem has been extended allowing entities to evolve over time after which they are clustered to form entity-dependent stories [AKS+14] and has also been studied for evolving correlation between textual data describing entities [CHSZ16]. This core idea is analogous to the idea of the evolution of events over time that we discuss in our event integration system, STORYPIVOT, but differs from our idea of incremental entity resolution because of the temporal aspect of the data. In contrast to previous work, the problem presented in STORYPIVOT is multi-faceted while previous research has mostly focused on entities only. Specifically, for a story to evolve, it is not only the entities but also the topics, sentiments etc. that may evolve.

Finally, an alternative to incremental ER which has also been published in 2014 is called progressive ER [AKM14] which aims to maximize the effectiveness of the incremental updates as early as possible. Instead of searching for an optimal solution, this line of work tries to find good enough ER solutions through effective reordering of computation.
Chapter 2: Background

2.1.3 Data Fusion

As mentioned previously, data fusion is the task of finding a common representation for a set of tuples. In our initial example for data integration, we were given several records describing countries and their capitals. The table contained multiple problems that a data fusion algorithm may encounter, for example did some of the columns contain the same value written differently and some contained different values in the same column for the same row values. Data fusion tries to resolve these inaccuracies for example following a set of rules as defined by a domain expert. An overview of these and other data fusion techniques is given in [BN09]. More recently, research on data fusion has moved away from static rules to incorporating machine learning in order to identify good fusion options. One of these research areas is to establish source accuracies and to use those to influence which data items are chosen for the fused object [DBS09a, RDGS15]. For example, if data source $s_1$ in Figure 2.1 is known to be reliable, while $s_3$ sometimes contains erroneous values, then the probability of data values in $s_1$ to be chosen for the fused object are higher for $s_1$ than $s_3$. These estimates may change over time which is the reason static rule-based approaches do not work well here.

Another problem that researchers have examined in this context is the problem of ‘copy detection’. That is, given a set of data sources, how can the system identify which data sources copy their content from each other. Copy detection is crucial to establish high quality data fusion results because copied values distort which data value is chosen. To address this problem, for example [DBS09b] proposes to track the coverage, exactness, and freshness of data sources and use machine learning techniques to establish copying relationships of records.

Data fusion is commonly applied on top of an ER solution as shown in the running example. It then unifies these entities. However, data fusion algorithms often do not modify the data values they are given. Instead, they pick one of the provided data values as representative. Thus, if the data contains errors, for example misspellings, formatting problems etc., data fusion will not be able to address them as is. As a result, many data integration systems deploy data cleaning algorithms either as part of the data fusion process or in combination with any of the previous integration steps. We discuss this area of research therefore next.
2.1: Automated Data Integration

2.1.3.1 Data Cleaning

Data cleaning is the task of finding syntactic and semantic inconsistencies in an input set of tokens, i.e., for example row entries in a column of a mediated schema table. Such inconsistencies may occur for a variety of reasons. As discussed for the example in the introduction of this dissertation (Figure 1.3), a few of these reasons are mismatches during the schema alignment phase, mistakes when the values were entered etc. Our work proposes new online mechanisms for executing semantic data cleaning as part of the integration process or interactively when the user edits her data. Therefore, we discuss two different cleaning techniques in detail next outlier detection and data repairs. Furthermore, we also show how data cleaning has been incorporated into other data integration operators in recent years.

Outlier Detection

Outlier detection is a topic that has been addressed in a variety of contexts, for example in bio informatics, data analysis, or clustering. Generally, outlier detection techniques belong to either of three categories: supervised, semi-supervised, and unsupervised [HA04]. Supervised outlier detection learns specific types of outliers using a training dataset. Once it has internally modelled these outlier types, it can be applied to new input data and will then judge whether there are outliers in this data that are similar to the ones observed in the training set. The problem with this kind of techniques is that if the training dataset is unbalanced, outlier detection results become unpredictable. As this occurs frequently [CJK04], semi-supervised outlier detection has become an alternative. Here, the system learns to expect ‘normality’ but is able to recognize ‘abnormal’ values. Abnormal values are classified as outliers. If they are tagged afterwards as normal, the model gets readjusted interactively. The last class of outlier detection algorithm is what our work focuses on: unsupervised outlier detection. In Section 3.2, we propose a novel unsupervised diagnostic outlier identification mechanism. More specifically, we introduce a density-based neighborhood outlier technique. Other techniques that belong to the same class of outlier detection mechanisms include distance metrics as for example used in (variations of) k-Nearest Neighbors (KNN) [Alt91, HKF04, SZK15]. Alternative unsupervised learning techniques include active learning [AZL06] and rule-based outlier detection [PCMHM16]. The former has been developed to address outlier detection as a classification problem similar to our multi-group clustering approach. In contrast, rule-based outlier detection
has been studied in the context of databases. Here, type-dependent metadata extension is applied to determine inconsistent records.

**Data Repairs**

Research on data repairs has mostly focused on repairing data with the help of functional dependencies (FDs). That is, if a (web) table is the input to the repair algorithm, it will examine the dependencies between the column values and then provide suggestions for data repairs. In [BFG+07] and later [CFG+07] the authors propose to use these dependencies as part of the data cleaning process. This line has been further extended by introducing the notion that not only the data may need to be repaired but also the constraints that allow us to judge whether a token should be repaired or not [CM11]. In our work, we shortly discuss token corrections which is analogous to the idea of data repairs where we leverage dependencies between tokens to allow us to determine good repair candidates. However, we do not rely on correlation between tabular values. Instead, we observe co-occurrences more broadly, for example in web tables or in enterprise data, and base our observations on set semantics. We thus develop a corpus of potentially correlated values which allows us to repair even a single-column table which is not possible with current FD-based approaches.

As an alternative to fully automated repair systems, ‘human-in-the-loop’ algorithms have gained prominence in this research area. For example, [YEN+11] proposes ‘Guided Data Repair’, a semi-automatic mechanism that implements continuous semi-automatic data repairs. Involving human feedback in the data repair cycle in general has led to a whole new research area often referred to as data wrangling. The idea of data wrangling is to use semantic information that provides analysts with targeted data repair options. These analysts are part of an interactive repair loop [KPHH11, SBI+13]. But not just the technical aspects of this human-system interaction and how the information gets propagated is crucial to the success of data wrangling approaches. Indeed, research has further explored how humans interact with such a system and which visual components can enhance the user’s ability to identify and correct erroneous data values [KHP+11].

**Combining Data Cleaning with Other Data Integration Operators**

Though data repairs and outlier detection are often applied as part of the data fusion process, data cleaning can also be interleaved with other data integration operations. For
example in [FMTY14], the authors show that entity resolution and data repairs can be combined to provide a consistent clustering of entities. Furthermore, there exists work that leverages pair-wise semantic consistency of columnar information extracted from web tables [CHCG15]. This idea is similar to FD-based data cleaning but here, correlation is used to determine how two (or more) tables can be integrated and aligned. As a follow-up, the authors suggest to use the same semantic information to establish relationships between domains with different syntactic characteristics [HGC15]. For the purpose of our work on online data cleaning with a given set of input tokens, this line of work is unfortunately not applicable as in all of them, cleaning the dataset while executing the operator is a side-effect rather than what the algorithms are designed to do. Thus, semantic or syntactic errors outside of the scope of the operator cannot be addressed correctly.

2.2 Crowdsourcing

We discuss related work in crowdsourcing in this section. As our work focuses on entity resolution with a dynamic crowd, we will explain alternative approaches in Section 2.2.3. However, there has been a recent surge in crowdsourcing research, thus we will give a short overview of relevant but not ER-centric research first after which we focus on ER techniques in a crowdsourcing context. First, we examine the area of crowd quality, i.e., how we can guarantee that the crowd answers as truthfully as possible. This is relevant to our work because one of the assumptions that we make is that the crowd may make mistakes. Second, crowdsourced operators are a related research area as ER is a crowdsourced operator that provides a data integration operation as other operators such as filter, join, or top-k do. Finally, we examine crowdsourced entity resolution in detail which is one of the research problems that this dissertation addresses.

2.2.1 Crowd Quality

Since defining and computing good similarity measures is not always possible, there has been further work [BIPR12, GDD+14, GWKP11, YJJJ12] that minimize the use of distance functions for record comparisons. These methods either rely on Bayesian modelling or similar mechanisms to approximate the answers of their data sources after extensive data collection or apply knowledge specific to a certain platform and its characteristics [BBCM13, IPW10, KBK+12]. Error intervals for crowd workers or error estimates
per worker group are alternative ways to model worker quality [JGP13]. In contrast, our method provides good ER results in the absence of similarity distances and is able to provide an ER result without any prior training of our mechanism. Information on the quality of crowd workers can be leveraged with our approach by requiring high quality workers, for example determined through their behavioral patterns [KKMF11], to answer the current top candidate pair. Thus, this type of research can be used to enhance the computed ER result. Nevertheless, noise in the answer set cannot be excluded categorically even if worker quality is high as these workers may make mistakes and provide erroneous information.

### 2.2.2 Crowdsourced Database Operators

There has been a lot of research on crowdsourced operators (filtering, top-k, and entity resolution) under the assumption of predefined error behavior of the crowd workers. This research can be divided into two categories: Approaches that rely on the crowd to give answers that can be monotonically aggregated [WKFF12, WMGM13] and those techniques that take specific error behavior of the crowd into consideration [DKMR13, GPG12, PGMP12]. Our approaches vary from the first group of algorithms as we tolerate and in fact embrace noisy behavior of the crowd. To the best of our knowledge, none of the techniques developed in the second group of algorithms can be used to resolve the entity resolution problem efficiently.

### 2.2.3 Crowdsourced Entity Resolution

Recently, hybrid human/machine entity resolution algorithms have attempted to automatically integrate humans as part of the ER process to increase the reconciliation quality [LCP13, YJJ12]. In [WKFF12] the authors combine automatic machine learning techniques with crowdsourcing, whereas [WLK13] extends the work to further reduce the cost of crowdsourcing by taking transitive relationships into account and also adjusting the crowdsourcing process according to automated similarity measurements. One of the assumptions that is commonly made in previous work is to consider worker quality as an orthogonal problem as well as not taking negative feedback from humans (i.e., that two entities do not match) into account when creating an ER solution [WLK13, WLGM13]. Thus, eventual conflicts in the crowdsourced comparisons are discarded or ignored. Fur-
thermore, there has been work on probabilistic crowdsourced ER [VG15] which proposes a maximum likelihood approach. However, the proposed strategy is NP-hard and thus infeasible to compute in an online setting. Next to interpreting the crowd worker’s feedback, research has asked the question what kind of feedback should be requested from the crowd in the first place [VBD14].

Although there exist a variety of approaches for crowdsourced ER, most of them leverage the same basic algorithmic model shown in Algorithm 1. Initially, they generate a set of record pairs (candidate pairs) \([r_i, r_j]\). This could be a complete set of record pairs or subset of those (for example using automatic similarity comparison to remove unlikely record pairs). These pairs are then sorted with regard to a predefined priority metric and added to a priority queue \(Q\) (Line 3). An example for such a priority queue is to order them according to their similarity in an automatic similarity computation. Iteratively, the top record pair is now retrieved and published as a task on a platform like Amazon Mechanical Turk to obtain information whether \(r_i\) and \(r_j\) in fact belong to the same entity (Line 6). Whenever a crowd worker responds, the answer is integrated into the votes graph \(G\). Based on the new information in \(G\), a clustering algorithm then determines the current entity resolution solution \(C\) (Line 9). We refer to this as the data interpretation problem. For example a simple clustering mechanism would be to merge all records \(r_i\) and \(r_j\) into the same real-world entity if the positive feedback of the crowd workers outweighs the negative feedback, i.e., if for the positive votes \(p\) and negative votes \(n\) the inequality \(p_{ij} > n_{ij}\) holds. Finally, the new information from the crowd may influence other record pairs, which can lead to an adjustment of the priority queue for record pairs (Line 11) and is also referred to as next-crowdsource problem. For example, if \(r_i\) and \(r_j\) as well as \(r_j\) and \(r_k\) are assigned to the same cluster, then asking for record pair \([r_i, r_k]\) is superfluous if the algorithm exploits transitivity.

Generally, research on entity resolution in the context of crowdsourcing adheres to the framework shown in Figure 2.2. The different research areas can be differentiated by how they address (a) the next-crowdsource problem and (b) the data interpretation problem. In the next-crowdsource problem, task ordering can process in a monotonic manner or alternatively already resolved pairs may be reconsidered if evidence points to a mistake in the previous decisions. This non-monotonic task ordering is based on the assumption that input information is unreliable. That is, to improve quality, it should be possible to re-evaluate the ER solution to correct the initial mistake of the workers. The second difference between different classes of crowdsourced ER is that they either leverage complete
Chapter 2:  Background

Algorithm 1. General ER algorithm for crowdsourced on-demand input.

1. $Q, C \leftarrow \emptyset$
2. // add elements to the queue
3. foreach $[r_i, r_j] \in R^2; i \neq j$ do
   4. $Q$.priorityAdd($[r_i, r_j]$)
4. // iteratively crowdsource and adjust clustering
5. foreach $[r_i, r_j] \in Q$ do
6.   $v_{ij} \leftarrow$ crowdsource($[r_i, r_j]$)
7.   // data interpretation problem
8.   C.update($v_{ij}$)
9.   // next-crowdsource problem
10.  Q.adjust(C)
11. return $C$

(positive and negative even if contradictory) votes from the crowd or request a consensus decision. As a result, there exist three different strategies in the solution space for crowdsourced ER: a) fault-tolerant exhaustive, b) fault-tolerant, and c) consensus-based strategies. Notice that combining non-monotonic and consensus-based mechanisms is not possible. The reason is that consensus mechanisms can never lead to contradictions in the ER solution, thus violating the non-monotonicity property. In the following, we examine how these categories of crowdsourced differ from each other and show the most prominent examples of existing work in either category.

Consensus-Based Entity Resolution

Entity resolution strategies that are based on consensus decision usually allocate a fixed repetition budget for each crowdsourcing task. The crowd worker’s answers are then aggregated by task and consolidated according to some previously defined ER algorithm such as transitive closure or sorted neighborhood as shown by CrowdER [WKFF12] or Whang et al [WLGM13]. Errors made by the crowd workers are thus masked in the hope that a sufficient number of repetitions will result in the correct answer per task. In fact, these strategies optimize for crowdsourcing cost under the assumption of a perfect crowd. Thus, consensus is always reached and there are no contradictions in the record pairs that
need to be resolved.

Fault-Tolerant Entity Resolution

We term fault-tolerant entity resolution those ER mechanisms that take as input all information made available by crowd workers and build an ER solution on top of that. In contrast to consensus-based ER, these ER mechanisms do not reject any of the crowd answers which introduces noise into the votes graph. The challenge is then to find a clustering on top of these possibly contradicting bits of information that maximizes the agreement between crowd answers. Our research, presented in Section 3.1.2, builds upon preliminary work published in a technical report [GNK+15] that first introduced the notion of fault-tolerant entity resolution. Related work in this category [VG15] has introduced a theoretic crowdsourced ER solution that uses maximum likelihood methodology to find the optimal ER solution. This methodology is equivalent to using correlation clustering and is shown to be NP-complete. For clustering, the authors therefore fall back onto spectral clustering and transitive closure as alternative ER mechanisms. Their work furthermore addresses the next-crowdsourcing problem by finding those tasks that have the highest projected impact on the entity resolution solution. This estimate extends prior work because it is not only based on the candidate positive crowd answers but also possibly negative responses. In that respect, it is similar to our work on the next-crowdsource problem although we
examine not only uncertainty reduction strategies but explore alternative means of optimizing for error reduction in the clustering solution.

**Fault-Tolerant Exhaustive Entity Resolution**

Exhaustive (non-monotonic) entity resolution differs from monotonic exploration of the ER space because record pairs that have been examined before can be re-evaluated at a later point in time. Thus, it is possible to reverse a decision once made during the clustering process. Prior work has not yet considered non-monotonic task execution for two reasons. First, it is more expensive especially if budget is invested on ‘hard’ tasks, i.e., tasks that crowd workers often disagree on. Second, existing work commonly reasons on a task level, i.e., the next question to ask is not one question for task but a set of questions for one task. As a result, once these questions have been asked, there exists sufficient signal from the crowd to determine an ER solution. To the best of our knowledge, our work is the first to explore non-monotonic task ordering in an extensive evaluation. In addition to sequential task execution, we also report on the trade-off between single-question execution and batch processing in Section 3.1.2 which impacts the output quality of non-monotonic as well as monotonic crowdsourced ER.

**2.3 Alternatives to Online Data Integration Systems**

The system that we develop in the second part of this dissertation, STORYPivot, is an online event integration system. This type of systems is related most closely to event processing systems which extract events from news articles to show trends or highlight when topics become ‘hot’. More generally, these systems can be seen as streaming systems that execute some specific integration task. Thus, we next describe work related to STORYPivot that has been done for general-purpose streaming systems and then focus on event processing systems in detail.

**Streaming Systems**

Streaming systems [BBD+02, Mut05] have been an increasingly important area of research over the last years because of the sheer amount of data that systems have to process in near real-time. Here, related work has focused on applying traditional database methodology
2.3: Alternatives to Online Data Integration Systems

such as joins and top-k processing on data streams [GÖ03, MAEA05]. In contrast, our work aims to solve the problem of continuously integrating and maintaining a large amount of data where the data may come in through data streams but needs to be integrated analogous to traditional data integration systems. At the same time, we allow the system to maintain multiple channels of communication similar to the execution of windowed joins on multiple streams simultaneously such as discussed in [GÖ03]. Aggregation in the context of streaming systems has been explored for example in [AM04, CM05, SKB16]. However, these techniques are most commonly used for estimating aggregation objects such as counts or sums.

Event Processing Systems

Repositories for real-world events such as GDELT [BFS11] or EventRegistry [LFBG14] store structured information on real-world events extracted from news articles in a variety of online newspapers. The goal of these systems varies from tracking political unrest to showing current and trending events. A popular use case for such systems is political science which tries to predict political unrest, crisis, and conflicts [KCKR14, SD14, WMD+13]. Furthermore, financial sciences have shown an increasing amount of interest in event processing systems because of their ability to help understanding and drawing connections between real-world events [Hin14]. Detecting events that have not been previously identified is another line of research that is relevant to our work. Here, research aims to characterize events [PBNG13] or to learn about existing events [AS12] to discover when new events occur. In contrast, the goal of STORYPIVOT is to show the evolution of stories over time to help users understand news articles and to let expert users explore stories within and across data sources. Furthermore, we discuss in our work how we can design event processing systems from scratch to be able to handle complex integration techniques efficiently and how these techniques are generally applicable for a variety of data integration systems and not only limited to event processing.
This chapter discusses the contribution of this dissertation to the methodology of data integration. That is, it evaluates new and improved entity resolution and data cleaning algorithms that handle data sources dynamically. We first discuss our findings for the task of entity resolution. This line of work can be split into advances in automated entity resolution and algorithms that handle crowdsourced information when generating an ER solution. Afterwards, we introduce a novel technique for leveraging the context of tokens in a dataset to enable semantic data repairs.
3.1 Entity Resolution

As described previously, entity resolution is a common task in the data integration process. It tries to determine whether a set of records point to the same real-world entity by computing the (syntactic and/or semantic) similarity between records. These similarities are then used to compute the entities. For the purpose of our work, we think of records as nodes in a graph. Edges between two nodes symbolize their similarity. Entities can then be computed as clusters where each node in the same cluster belongs to the same real-world entity as perceived by the clustering algorithm. Such clustering algorithms have been introduced in Section 2.1.2.2.

In the first part of our work, we have examined these existing clustering algorithms that were designed for batch processing records. To enable them to handle dynamic workloads, we propose mechanisms to adapt them to fast-paced data without loss of quality. Observing that all of these clustering techniques optimize for a global objective function, we then propose a greedy technique that instead executes local optimizations. We experimentally evaluate all of our approaches extensively.

The second part of our work on entity resolution focuses on finding entities based on crowdsourced data. We especially focus on how we can enable fault-tolerance in our algorithms, i.e., being able to handle erroneous answers from the crowd, and how we can minimize the number of times information is retrieved from the crowd. For that purpose, we explore first graph-based traversal algorithms that enable us to compute the likelihood that two records belong to the same entity. Using this likelihood and different strategies of accessing the crowd, we show the impact of erroneous crowd answers on a relatively simple task such as ‘Is A equivalent to B?’ experimentally.
3.1: Entity Resolution

3.1.1 Incremental Entity Resolution

In this section, we discuss the adaptation of static batch algorithms to allow for dynamic data management following Problem 1.1.1 introduced in Section 1.1.1.1 which has been published in [GDS14]. The goal of incremental entity resolution is to be able to quickly update an ER solution if the underlying recordset changes. That is, given a set of records and its corresponding clustering, we want to enable the ER framework to recompute only those parts of the clustering that have been modified by the changes to the records instead of rerunning a batch ER algorithm. We want to achieve two goals: First, we want the incrementally computed ER solution to have the same result quality as its corresponding batch variant. Second, we want the partial recomputation to be significantly faster than the batch computation.

Prior to our work, incremental graph clustering was achieved by either merging a new record with an existing cluster or generating a new cluster for that record. In other words, we assume a new record either refers to an existing entity or a new entity. The terms entity and cluster will thus be used interchangeably in the following. Merge-only strategies do not give theoretical guarantees about the quality of the entities in the solution. In fact, if due to mistakes made during the similarity computation or missing information a record is wrongly assigned to a cluster, these strategies are not able to fix their erroneous clustering. As an illustration for this problem, take the following example.

**Example 3.1.1 (Motivation).** Going back to the introductory example for entity resolution in Figure 1.1 (a), we see a recordset that represents five businesses through records $r_1 - r_{10}$.
Pair-wise similarities between these records are computed using the columnar values in the following columns: (1) name, (2) street address excluding house number, (3) house number in street address, (4) city, and (5) phone. The similarity is set to 1 if all values are the same, 0.9 if four of them overlap, 0.8 if there is an overlap between three values, and 0 otherwise. The corresponding similarity graph and the clustering with correlation clustering as an objective function is shown in Figure 1.1 (b). Through mistakes in the input data (wrong phone number), \( r_4 \) is merged with \( r_1 - r_3 \). Furthermore, \( r_5 \) and \( r_6 \) are in separate clusters because \( r_6 \) is missing some columnar values. Finally, \( r_7 - r_9 \) are merged even though \( r_9 \) and \( r_{10} \) belong to the same entity because \( r_9 \) seems similar to these records.

Using a merge-only incremental strategy will not resolve these issues if more data is added to the recordset. For example, take the updates described in Figure 1.2 which insert records \( r_{11} - r_{17} \). Applying the naive merge-only strategy, we obtain the clustering shown in Figure 3.1. It does not correct the issues pointed out earlier but instead assigns each record to an existing cluster except for \( r_{11} \) which is assigned to a singleton cluster. A now correct clustering which is possible due to the addition of the records is shown in Figure 1.2 (b). The goal thus has to be to enable the framework to reach this clustering instead, i.e., to always provide the best possible clustering.

Related work as discussed in Section 2.1.2 does not provide means to enable high quality ER solutions for fast-changing recordsets. To the best of our knowledge, this is the first research work that shows that we can modify existing algorithms to facilitate near real-time propagation of changes in the recordset to the ER solution while maintaining equivalent quality as if we had computed the ER solution from scratch. The work presented in the next sections thus makes the following contributions.

**Optimal Incremental Graph Clustering.** We propose and describe two optimal modifications to batch graph clustering algorithms that apply the objective function of the batch algorithm on a subset of the data.

**Greedy Incremental Graph Clustering.** We approximate our findings in a greedy approach that mimics the behavior of the optimal algorithms but follows a local rather than global optimization scheme.

**Implementation Details for Correlation Clustering and DB-Index.** We describe in detail how these algorithms can be used for two well-known ER algorithms, cor-
relation clustering and DB-index, and verify their applicability in an extensive experimental evaluation.

The approaches that we present are not only limited to the two clustering algorithms that we discuss. Instead, they can be applied to all graph clustering techniques where the graph has records as nodes and edges in the graph are the similarity between two nodes. We next introduce the formal problem statement of incremental entity resolution after which we present the optimal incremental ER techniques. Using these, we develop a greedy technique and show how all approaches can be mapped to correlation clustering and DB-Index.

3.1.1.1 Problem Overview

In this section, we discuss the formalized problem statement of incremental entity resolution. It focuses on one of the components of the ER process that we discussed in Section 2.1.2, graph clustering. That is, given a set of records $D$, entity resolution becomes a clustering problem on top of a similarity graph as described previously. We denote the clustering of $D$ with $L_D$ in the following. Given a batch clustering function $F$, we obtain $L_D$ by applying $F$ on $D$, i.e., $F(D) = L_D$. Ideally, $L_D$ has high precision as well as high recall meaning that records in the same cluster in fact refer to the same entity and that all records that refer to the same entity are in the same cluster.

Modifications of the recordset can be three-fold. First, insert operations add records to $D$. Second, the delete operation removes an existing record from $D$. Finally, a change of a record modifies one or multiple existing records. The change operation can also be realized by the combination of a delete and insert operation. We refer to a set of modifications, i.e., a bundle of insert, delete, or change operations, as an increment which we denote as $\Delta D$. The combination of the existing dataset $D$ and an increment is referred to as $D + \Delta D$. As such an increment may contain delete operations, the number of records in $D + \Delta D$ may be smaller or equal to the number of records in $D$ which means that $|D + \Delta D| \leq |D| + |\Delta D|$ must hold. A valid operation in an increment then can be defined as a) an insert operation of a record that did not previously exist, or b) a delete or change operation of a record that exists in $D$. Formally, we define incremental entity resolution as follows.

**Definition 3.1.1 (Incremental Entity Resolution)**. Let $D$ be a set of records and $\Delta D$ be an increment to $D$. Let $L_D$ be the clustering of records in $D$. Incremental entity resolution
clusters records in $\mathbf{D} + \Delta \mathbf{D}$ based on $\mathcal{L}_\mathbf{D}$. We denote the incremental entity resolution method by $f$, and denote the results by $f(\mathbf{D}, \Delta \mathbf{D}, \mathcal{L}_\mathbf{D})$.

To iterate, incremental entity resolution has two goals. On the one hand, it aims to improve performance significantly compared to its corresponding batch clustering algorithm especially if the increment is small. Specifically, the computation of $f(\mathbf{D}, \Delta \mathbf{D}, \mathcal{L}_\mathbf{D})$ should be faster than the computation of $F(\mathbf{D} + \Delta \mathbf{D})$ if $|\Delta \mathbf{D}| \ll |\mathbf{D}|$ holds. On the other hand, incremental entity resolution should achieve equivalent similarity to its reference batch algorithm. We denote this constraint as $f(\mathbf{D}, \Delta \mathbf{D}, \mathcal{L}_\mathbf{D}) \approx F(\mathbf{D} + \Delta \mathbf{D})$ where $\approx$ is a comparison function based on the precision and recall of the clustering output.

**Example 3.1.2 (Incremental Execution).** Going back to the running example of business processes, assume that $\mathcal{L}_\mathbf{D}_0$ refers to the clustering shown in Figure 1.1 (b) which is a clustering based on records $\mathbf{D}_0 = \{r_1 - r_{10}\}$. Given increments $\Delta \mathbf{D}_1 - \Delta \mathbf{D}_4$ shown in Figure 1.2 (a), we can now iteratively apply these increments. The final clustering contains six clusters as shown in Figure 1.2 (b). It is the same clustering that we would achieve applying a batch algorithm on the combined dataset $\Delta \mathbf{D}_0 - \Delta \mathbf{D}_4$ and should be generated by the incremental entity resolution algorithm as well.

As this problem is a graph clustering problem, we now introduce our notation for the similarity graph $G(V, E)$ which contains as nodes the records in $\mathbf{D}$. Specifically, every node $v_r \in V$ is mapped to a a record $r \in \mathbf{D}$. An edge $(v_r, v_{r'}) \in E$ with weight $\text{sim}(r, r')$ then denotes the similarity of records $r$ and $r'$ where $r, r' \in \mathbf{D}$ holds and their similarity $\text{sim}(r, r')$ is in $[0,1]$. We say that $r$ and $r'$ are identical if $\text{sim}(r, r') = 1$ and distinct if $\text{sim}(r, r') = 0$. A clustering on top of $G$ is denoted as $\mathcal{L}_G$. Note that the notion of the clusters found in $\mathcal{L}_G$ is equivalent to the notion of entities in $\mathcal{L}_\mathbf{D}$. Updates can change $\mathcal{L}_G$ as follows.

**Insert Operation.** Adding a record to $\mathbf{D}$ is the same as adding a node to $G$.

**Delete Operation.** Removing a record from $\mathbf{D}$ is equivalent to the removal of the corresponding node from $G$.

**Update Operation.** Changing a record in $\mathbf{D}$ means that the existing edges have to be removed from $G$ and all new edges have to be added to the graph.

Changes to the graph structure, i.e., an increment to the graph, is denoted as $\Delta G$ in the following. If a graph $G$ is expended with $\Delta G$, the resulting graph is $G + \Delta G$. Furthermore,
we denote the new clustering of $G$ as $L_G$ and the incremental entity resolution process as $f(G, \Delta G, L_G)$.

### 3.1.1.2 Optimal Incremental Entity Resolution

In this section, we first introduce desirable properties for objective functions that can be used for incremental entity resolution. We then present two algorithms that solve the incremental ER problem optimally with respect to a reference objective function. We show by the example of correlation clustering and DB-index clustering how the different properties of an objective enable or disable our methods.

#### Desirable properties of linkage

To enable incremental entity resolution, an objective functions should have certain desirable properties. A (subset of) these properties can be found in most commonly applied graph clustering techniques that are used for entity resolution. We shortly introduce each of these properties and explain their purpose for ER. To formally reason about their properties, we use $O$ as the notation for an objective function in graph clustering. To find the optimal clustering solution, we minimize $O$ analogous to minimizing a specific penalty function. Our line of argumentation is equally applicable when maximizing $O$.

The optimal clustering of $G$ using $O$ is then denoted as $L_G^{O,\text{opt}}$.

**Connectivity.** The optimization score of $O$ is dependent on the edge similarity between the records within a cluster. That is, if two records $r$ and $r'$ are dissimilar, i.e., $\text{sim}(r, r') < 0.5$, it is intuitive that putting them into the same cluster increases $O$. The connectivity property is based on this observation. It is defined as follows.

**Definition 3.1.2** (Connectivity). Let $L_G$ be a clustering of $G$ and $L_G'$ be a clustering obtained by putting two disconnected clusters in $L_G$ into the same cluster. We say $O$ satisfies this property if for every such $L_G$ and $L_G'$, $O(L_G) < O(L_G')$.

In words, connectivity implies that the penalty of a modified clustering $L_G'$ which contains disconnected records from two originally separated clusters should be higher than the original clustering $L_G$ which kept these records separated.
Chapter 3: Data Integration Methodology

Figure 3.2: Desirable entity resolution properties.

**Locality.** This property is stricter than connectivity and if an objective function has this property, it automatically has the connectivity property as well. Locality is formally defined as follows.

**Definition 3.1.3 (Locality).** Let $G_1$ and $G_2$ be a split of $G$ such that there is no edge between $G_1$ and $G_2$ (Figure 3.2(a)). We say $O$ satisfies this property if for every such $G$, $G_1$, and $G_2$, $L_{G_1}^{O,\text{opt}} \cup L_{G_2}^{O,\text{opt}}$ forms an optimal clustering for $G$ under $O$.

This property implies that if $G$ contains multiple connected components, these components can be put into separate subgraphs $G_i$. Combining the optimal clusterings of all of these $G_i$, $\bigcup L_{G_i}^{O,\text{opt}}$, is then equivalent to $L_G^{O,\text{opt}}$. It means if there is no direct or indirect link between any two subgraphs, their corresponding optimal clusterings should be independent.

**Exchangeability.** While locality and connectivity are based on disconnected subsets of $G$, exchangeability focuses on the clustering properties within a connected component. Specifically, it says that if there exists a subgraph $G' \subseteq G$ that can be connected to the remaining parts of $G$ and contains a subset of clusters $\bar{C} \subseteq L_G^{\text{opt}}$, then replacing the optimal clustering for $G'$ means that $G$ also maintains an optimal clustering. Exchangeability is feasible because some objective functions may have several clusterings that achieve the same penalty. Thus, interchanging (a subset) of these clusterings should not modify the overall penalty.

**Definition 3.1.4 (Exchangeability).** Let $\bar{C} \subseteq L_G^{\text{opt}}$ be a subset of clusters and $G' \subseteq G$ be the subgraph containing only nodes in $\bar{C}$ and edges between them (Figure 3.2(b)). We say $O$ satisfies this property if for every such $G$ and $\bar{C}$, $\bar{C}$ is an optimal clustering for $G'$ and replacing $\bar{C}$ with any other optimal clustering of $G'$ obtains an optimal clustering for $G$ under $O$. 

42
Separability. The records in a connected component within $G$ are not necessarily directly connected with each other. Instead, chain connected components may contain records that belong to different clusters. Separability defines the penalty behavior of these as follows.

**Definition 3.1.5 (Separability).** Let $G_1, G_2, G_3$ be a partition of $G$ such that (1) $G_1$ and $G_3$ are disconnected; (2) there exists an optimal clustering for $G_1 \cup G_2$ with no cluster across $G_1$ and $G_2$; and (3) there exists an optimal clustering for $G_2 \cup G_3$ with no cluster across $G_2$ and $G_3$ (Figure 3.2(c)). We say $O$ satisfies this property if for every such $G, G_1, G_2$ and $G_3$, there exists an optimal clustering for $G$ with no cluster across two or three of the subgraphs $G_1 - G_3$ under $O$.

Monotonicity. The last desirable property for graph clustering functions is independent of the previous four properties. Monotonicity defines how changes in the pair-wise similarities affect the optimal clustering.

**Definition 3.1.6 (Monotonicity).** Let $v_1, v_2 \in V$ be two nodes in the same cluster in $\mathcal{L}_{G_{\text{opt}}}^O$. Let $G'$ be a graph obtained by increasing the weight of edge $(v_1, v_2)$ in $G$. We say $O$ satisfies positive monotonicity if for every such $G$ and $G'$, $\mathcal{L}_{G_{\text{opt}}}^O$ is also an optimal clustering of $G'$. Let $v_1, v_2 \in V$ be two nodes in different clusters in $\mathcal{L}_{G_{\text{opt}}}^O$. Let $G'$ be a graph obtained by decreasing the weight of edge $(v_1, v_2)$ in $G$. We say $O$ satisfies negative monotonicity if for every such $G$ and $G'$, $\mathcal{L}_{G_{\text{opt}}}^O$ is also an optimal clustering of $G'$. We say $O$ satisfies the monotonicity property if it satisfies both positive and negative monotonicity.

Specifically, the monotonicity property states that if the two records are in the same cluster according to $\mathcal{L}_{G_{\text{opt}}}^O$ and their similarity is strengthened, the optimal clustering will not change. Analogously, if two records $r$ and $r'$ are in separate clusters and their similarity decreases, $\mathcal{L}_{G_{\text{opt}}}^O$ will not be different than $\mathcal{L}_{G_{\text{opt}}}^O$.

Relationship between these Properties. The relationship between these five properties is shown in Figure 3.3. Formally, these relationships are stated in this theorem.

**Theorem 3.1.1.** For the relationships between the properties, a) locality implies connectivity; b) exchangeability implies locality; c) separability implies locality; d) there exists an objective function that satisfies exchangeability but not separability and vice versa; e) there exists an objective function that satisfies connectivity but not monotonicity and vice versa.

How locality implies connectivity was explained previously. At the same time, locality as well as connectivity are implied by either exchangeability and separability. The reason
Chapter 3: Data Integration Methodology

Figure 3.3: Relationships between the properties.

is that both work on connected components which is a tighter bound than disconnected components. At the same time, exchangeability does not necessarily imply separability (and the other way around) because they are separate constraints on the structure of the optimal clustering. Finally, monotonicity is different than the other four properties as it focuses on the edge values rather than the clustering of nodes. As a result, there can exist a clustering that satisfies only monotonicity or a combination of monotonicity and all other properties (taking into consideration their dependencies).

Correlation Clustering & DB-Index Properties

We now show how these properties relate to correlation clustering as well as DB-index clustering. For DB-index clustering, we will show by example that it violates the aforementioned properties while we can proof that they hold for correlation clustering. This is partially due to the fact that both techniques refer to different clustering types, i.e., distance-based and adjacency-based clustering. Though we can guarantee the optimality of our incremental techniques only for those objective functions that have these properties, we show experimentally later that our incremental mechanisms nevertheless work well in practice in various settings.

Correlation Clustering. We now proof that the five properties hold for correlation clustering, i.e., that the following theorem holds.

**Theorem 3.1.2** (Correlation Clustering Properties). The five properties connectivity, locality, monotonicity, exchangeability, and separability are satisfied by the objective function of correlation clustering as shown in Section 2.1.2.2.
3.1: Entity Resolution

Proof. We separate the proof by property:

Connectivity. Clustering $\mathcal{L}_G$ has a lower penalty than $\mathcal{L}_G'$ because $G'$ contains two disconnected clusters in one cluster. Thus, $\mathcal{L}_G$ is considered the better clustering.

Locality. With $G$ as a similarity graph and $G_1$ and $G_2$ its two disconnected subgraphs, we say that the optimal clusterings of $G_1$ and $G_2$ result in the optimal clustering of $G$. Assume that there exists an even better clustering $\mathcal{L}_G^{\text{opt}}$. This clustering can only exist if for either $G_1$ or $G_2$ their optimal clustering is in fact not optimal. This is a contradiction.

Exchangeability. This property states that a set of clusters $\bar{C}$ that is derived as a subset of $\mathcal{L}_G^{\text{opt}}$ can be replaced by another optimal clustering of its corresponding subgraph $G'$ which then results in a new optimal overall clustering. Assume that we replace $\bar{C}$ with $\bar{C}'$, a better clustering than $\bar{C}$. Thus, $G'$ now has a lower penalty than before while $G \setminus G'$ has the same penalty. Then, $\mathcal{L}_G^{\text{opt}}$ is not optimal anymore which violates its definition.

Separability. Following the connectivity property, no cluster in $\mathcal{L}_G^{\text{opt}}$ can cross $G_1$ and $G_3$ if they are disconnected. We thus need to show that there exists an optimal clustering that does not have any cluster across $G_1 - G_3$. Using exchangeability, we can then show that there exists no cluster across $G_1$ and $G_2$ or $G_2$ and $G_3$.

Assume that there is a cluster $C$ in $\mathcal{L}_G^{\text{opt}}$ across $G_1 - G_3$ as described in Definition 3.1.5. It contains three clusters, $C_1 \in G_1$, $C_2 \in G_2$, and $C_3 \in G_3$, such that $C = C_1 - C_3$.

If we find a new clustering $\mathcal{L}$ that replaces $C_1 - C_3$ such that there are two clusters across the borders of the partitioned graphs, we show that separability does not hold for this graph. As $G_1$ and $G_3$ are disconnected, separating $C_1$ and $C_3$ incurs no penalty. However, we refer to $p_{12}$ as the penalty between $C_1$ and $C_2$ and $p_{23}$ is the penalty for not having a cluster across the $G_2 - G_3$ graph edge. Thus, the correlation penalty of $C$ is $p = p_{12} + p_{23}$.

Assume that there exists a supercluster $C_1'$ of $C_1$ that contains a subset of the nodes in $C_2$ (transforming $C_2$ to $C_2'$) and crosses the edge of graphs $G_1$ and $G_2$. Given the optimal clustering $\mathcal{L}_{12}^{\text{opt}}$ of $C_1$ and $C_2$ and the new clustering $\mathcal{L}_{12}$, we know that the penalty of $C_1'$ and $C_2' \in \mathcal{L}_{12}$ has to be less or equal to $\mathcal{L}_{12}^{\text{opt}}$. Thus, $p_{12} \leq |C_1||C_2| - p_{12}$ holds. Analogously, we can show the penalty computation of $G_2$ and $G_3$ and therefore $p_{23} \leq |C_2||C_3| - p_{23}$. Given that $G_1$ and $G_3$ are disconnected, we get as penalty for $\mathcal{L}$
Chapter 3: Data Integration Methodology

the following equation: $p' = |C_1||C_2| - p_{12} + |C_2||C_3| - p_{23} + |C_1||C_3|$. Evaluating our previous observations about the cross-cluster penalty scores, we see now that $p \leq p'$ must hold and thus separability holds for $G$.

**Monotonicity.** Assume that there exist two records $r_1$ and $r_2$ and their corresponding nodes $v_1$ and $v_2$. If their edge weight $(v_1, v_2)$ increases from $w_1$ to $w_2$ then the penalty $p_{opt}$ of $L_{opt}'$ increases to $p' = p_{opt} + w_1 - w_2$ on the modified graph $G'$. If $v_1$ and $v_2$ are in the same cluster in $L$ an increase in the edge weight signifies a smaller cohesion penalty which means $p' < p_{opt}$. This proves positive monotonicity. Analogously, if $v_1$ and $v_2$ are in different clusters and their edge weight decreases, $p'$ will decrease according to correlation clustering as the penalty for a less weighted cut edge is lower than that of the original edge. This proves negative monotonicity.

DB-Index. In contrast to correlation clustering, DB-index does not adhere to these properties as we will show in the following.

**Theorem 3.1.3 (DB-Index Properties).** The five properties connectivity, locality, monotonicity, exchangeability, and separability are not satisfied by the objective function of DB-index as shown in Section 2.1.2.2.

**Proof.** Again, we split our line of argumentation following the different properties. As a reference clustering for DB-index, we use Figure 3.4.

**Connectivity and Locality.** Consider Figure 3.4 where cluster $C_0$ contains two disconnected subgraphs $C'_0$ and $C''_0$, where $C'_0$ has $m$ nodes. Let $G$ be a graph that contains $C_0 - C_n$ and $G'$ a graph containing $C'_0 \cup C''_0 \cup k \in [1,...,n] C_k$. If connectivity holds, then the penalty of $L_{opt}'$ is lower than the penalty of $L_G$. According to the DB-index objective function, the intra-cluster distance of $C_0$ is $1 - \frac{0.5 + m(m-1)/2 + 0.5}{(m+2)(m+1)/2} = 0.5 + \frac{2m}{(m+2)(m+1)}$. Given an arbitrary large number of $m$, the distance is close to 0.5 and we denote it as $0.5 + \delta$. Here, $\delta$ is a positive number close to 0. The intra-cluster distance of any of the other clusters $C_k$, $k \in [1,n]$ then is $1 - (0.5 + 2\delta) = 0.5 - 2\delta$. For $C_0$ and $C_k$, the inter-cluster distance is 1 while the inter-cluster distance between $C_k$ and another $C_k'$, $k' \in [1,n]$ with $k \neq k'$, is $1 - \frac{12\delta}{1-\delta}/4 = \frac{1-4\delta}{1-\delta}$. For simplicity, we assume $\alpha = \beta = 0$ in the following but our conclusions hold for other choices for $\alpha$.
3.1: Entity Resolution

and \( \beta \) as well. The separation measure of the DB-index objective function is then
\[
M(C_0) = \frac{0.5 + \delta + 0.5 - 2\delta}{1} = 1 - \delta
\]
for \( C_0 \) and \( M(C_k) = 1 - \delta \) for any \( C_k \). As a result, the DB-index for \( \mathcal{L}_G \) is \( 1 - \delta \).

The split cluster in \( G' \) results in an intra-cluster distance for \( C'_0 \) and \( C''_0 \) respectively of 0.5. Their separation measure is 1. For \( C_k \), the separation measure remains the same as before. The DB-index for \( G' \) can then be computed as
\[
\frac{1 + 1 + (1 - \delta)n}{n+2} = 1 - \frac{n\delta}{n+2} > 1 - \delta.
\]
This indicates that \( \mathcal{L}_G \) is a better clustering choice than \( \mathcal{L}'_G \). As \( C'_0 \) and \( C''_0 \) are disconnected, DB-index thus violates connectivity and locality.

**Exchangeability.** The optimal clustering according to DB-index of Figure 3.4 is \( C_0 - C_n \). According to exchangeability, if we consider the subgraph \( G' \) that contains \( C_0 \), then the \( C_0 \) should be an optimal clustering for \( G' \). However, as we have shown above, \( \{C'_0, C''_0\} \) is the optimal clustering. This means that DB-index violates the exchangeability property.

**Separability.** To show separability, we split \( G \) into \( G_1 = \{C'_0\} \), \( G_2 = \{C_1, \ldots, C_n\} \), and \( G_3 = \{C''_0\} \). The definition holds as \( G_1 \) and \( G_3 \) are disconnected, an optimal clustering for \( G_1 \cup G_2 \) is \( \{C'_0, C_1, \ldots, C_n\} \), and an optimal clustering for \( G_2 \cup G_3 \) is \( \{C''_0, C_1, \ldots, C_n\} \). However, the optimal clustering for \( G \) following DB-index contains

![Figure 3.4: Violation of properties through DB-index.](image-url)
Chapter 3: Data Integration Methodology

\( C_0 \) which again encloses two disconnected clusters, \( G_1 \) and \( G_3 \). This violates the definition of separability.

**Monotonicity.** If we increase the weight \( w_1 \) of an intra-cluster edge to \( w_2 \) of a cluster \( C_k \), we know that \( D(C_k) \) decreases. That means in return that the separation measure \( M(C_k, C_{k'}) \) with \( k \neq k' \) decreases if \( D(C_{k'}) \) and \( D(C_k, C_{k'}) \) are kept the same. As \( M(C_k) \) is defined as the maximum of all \( M(C_k, C_i), i \neq k \), may not decrease if \( i \neq k' \) holds. In that case, the penalty would remain the same which violates monotonicity.

\[ \square \]

**Connected Component Algorithm**

The first approach that we present for incremental entity resolution is based upon the locality property. It is intuitive that if there exists no direct or indirect relationship before or after an increment between any two records \( r \) and \( r' \), a change to either should not influence the ER clustering of the other. We thus base our algorithm on the notion of a ‘connected component’, i.e., the set of directly or indirectly connected records. It is formally defined as follows.

**Definition 3.1.7 (Connected component).** Let \( G \) be a similarity graph and \( \Delta G \) be an increment on \( G \). We define the transitive closure of a node as the connected subgraph in \( G + \Delta G \) including the node. We define the transitive closure of an edge as the connected subgraph in \( G + \Delta G \) including the edge and its two nodes. The connected component of \( \Delta G \), denoted by \( T(\Delta G) \), contains the union of the transitive closures for each inserted, deleted, or changed node or edge.

Using this definition, we can now introduce a straightforward incremental approach that leverages the notion of a connected component. Given \( G \), \( \Delta G \), and the current (optimal) clustering \( L^{opt}_G \), the connected component approach, CONNECTED, proceeds in three steps.

1. It determines the connected component \( T(\Delta G) \) as defined in Definition 3.1.7.
2. It determines the optimal clustering \( L^{opt}_{T(\Delta G)} \), i.e., the optimal clustering of a batch objective function \( F \) on the transitive closure of the graph increment.
3. It then combines \( L^{opt}_{T(\Delta G)} \) with \( L^{opt}_G \) by replacing the old clusters for all nodes that are in \( \Delta G \) and adding those clusters for nodes that were previously not in \( G \).
Theorem 3.1.4 (Optimality of \textsc{Connected}). Algorithm \textsc{Connected} is optimal if and only if locality holds.

Proof. Assume that graph $G$ is split into disconnected subgraphs such that $G_1, G_2 \subset G$. If locality holds, then the union of their optimal clustering $\mathcal{L}_{G_1}$ and $\mathcal{L}_{G_2}$ results in an optimal clustering $\mathcal{L}_G$. \hfill \qed

To improve the connected component approach, we can further use a variant called the \textit{monotone connected component} approach or short \textsc{MonoConnected}. This approach examines the current clustering and ignores edges that a) are intra-cluster edges and increase in its edge weight or b) are inter-cluster edges and their weight decreases within the connected component. Thus, if a node is deleted, this approach will also ignore its edges to other clusters. The resulting connected component is then referred to as monotone because the monotonicity property holds.

Definition 3.1.8 (Monotone connected component). Let $G$ be a similarity graph and $\mathcal{L}_{G}^{opt}$ be the given optimal clustering on $G$. Let $\Delta G$ be an increment on $G$. The monotone connected component of $\Delta G$, denoted by $\hat{T}(\Delta G)$, is defined as follows.

- For each inserted node $v \in \Delta G$, $\hat{T}(\Delta G)$ contains its transitive closure.
- For each deleted node $v \in \Delta G$, $\hat{T}(\Delta G)$ contains its cluster in $\mathcal{L}_{G}^{opt}$ but does not contain $v$ and edges to $v$.
- For each edge $e \in \Delta G$ with increased weight, if $e$ is across clusters in $\mathcal{L}_{G}^{opt}$, $\hat{T}(\Delta G)$ contains its transitive closure.
- For each edge $e \in \Delta G$ with decreased weight, if $e$ is within a cluster in $\mathcal{L}_{G}^{opt}$, $\hat{T}(\Delta G)$ contains its transitive closure.

Example 3.1.3 (Connected Component Approaches). Going back to the running example, consider increment $\Delta D_4$ in Figure 1.2(a). Here, $r_{16}$ and $r_{17}$ as well as their associated edges are added to the similarity graph. With $T(r_{16}) = r_7 - r_{10}$ and the same transitive closure for $r_{17}$, the connected component contains records $r_7 - r_{10}, r_{16}$, and $r_{17}$. The optimal clustering for these nodes is shown in Figure 1.2(b). It contains clusters $C'_{4}$ and $C'_{5}$ which replace $C_4$ and $C_5$.

To show the applicability of \textsc{MonoConnected}, imagine next an increment $\Delta D_5$ which removes node $r_4$. The transitive closure of $r_4$ contains all nodes in $C_1'$ whereas $C_1''$ consists
only of $r_4$ and is thus not part of the closure. With CONNECTED, we would thus examine the nodes in $C'_1$ without modifying the clustering. In contrast, as all weight changes are monotonous, MONOCONNECTED would simply remove $C''_1$ without evaluating $C'_1$.

Given this stricter version of CONNECTED, we next show its optimality with respect to the original CONNECTED technique.

**Theorem 3.1.5 (Optimality of MONOCONNECTED).** Algorithm MONOCONNECTED is optimal if and only if locality and monotonicity hold.

**Proof.** We have already shown that CONNECTED is optimal so we now need to show that the subgraph of a connected component does not change under the monotonicity property.

We consider two cases. First, an edge has increased its weight and is an intra-cluster edge. Given positive monotonicity, this change will not modify the optimal clustering. Second, an inter-cluster edge may decrease its weight. Again, referring to negative monotonicity, the optimal clustering will not change.

**Proposition 3.1.1 (Complexity of CONNECTED and MONOCONNECTED).** Let $c(|G|)$ be the complexity of finding the optimal clustering on $G$. The complexity of CONNECTED and MONOCONNECTED is $O(c(|G + \Delta G|))$.

If $G + \Delta G$ is a fully connected graph, $T(\Delta G)$ and $\hat{T}(\Delta G)$ are the same as $G + \Delta G$ in the worst case. For this use case, both approaches would thus have the same complexity as their corresponding batch linkage algorithm. However, if $G$ consists of (a large number of) connected components, the transitive closure of these approaches may be significantly smaller. As a result, they provide the same result quality while having (significantly) better performance.

**Iterative algorithm**

Both of the approaches presented previously, CONNECTED and MONOCONNECTED, examine a subgraph of the data. However, in a worst case scenario, even this subgraph may be substantially sized. To further optimize our incremental technique, we thus limit the search space further, focusing on closely connected subgraphs next. Specifically, we now introduce an iterative algorithm, ITERATIVE, that initially considers the directly connected subgraph of an increment. This subgraph is referred to as the directly connected
3.1: Entity Resolution

component. I then iteratively widens the subgraph if the clustering has changed. We formally define a directly connected component as:

**Definition 3.1.9** (Directly connected component). Let $G$ be a similarity graph and $L_{opt}^G$ be the given optimal clustering on $G$. Let $\Delta G$ be an increment on $G$. The directly connected component of $\Delta G$, denoted by $\bar{T}(\Delta G)$, is then defined as:

- For each inserted node $v \in \Delta G$, the directly connected component contains $v$ and its connected clusters in $L_{opt}^G$.
- For each deleted node $v \in \Delta G$, $\bar{T}(\Delta G)$ contains its cluster in $L_{opt}^G$, but does not contain $v$ and edges to $v$.
- For each edge $e \in \Delta G$ with increased weight, if $e$ is across clusters $C_1, C_2 \in L_{opt}^G$, $\bar{T}(\Delta G)$ contains $C_1$ and $C_2$.
- For each edge $e \in \Delta G$ with decreased weight, if $e$ is within a cluster $C \in L_{opt}^G$, $\bar{T}(\Delta G)$ contains $C$.

The Iterative algorithm then iteratively proceeds as follows.

1. To initialize the execution, the algorithm determines all modified clusters and puts them into a queue $Q$.
2. For each queued cluster $C \in Q$, we then determine the directly connected component $G'$. On top of this, the optimal clustering is computed using the batch objective function $F$. If the clustering changes, i.e., $L_{opt}^G \neq L_{opt}^{G'}$, we determine the modified clusters $C'$.
3. All modified clusters are added to $Q$ if they are not already enqueued.
4. Jump back to 2. Repeats until $Q$ is empty.

**Example 3.1.4** (Iterative Approach). In Figure 3.5, we depict a scenario where Connected and MonoConnected potentially incur additional overhead while Iterative effectively limits the incremental computation space. Specifically, we observe that the directly connected component here contains $v \cup C$ while the connected component is significantly larger. If $v$ remains in a singleton cluster (it is not well-connected with $C$), the input recordset for $F$ is thus substantially decreased in size.
However, the iterative algorithm is not always better than the connected component approaches. Specifically, it may incur additional overhead when iteratively ‘bubbling’ through the connected component. Consider increment $\Delta D_4$ in Figure 1.2. After the initialization, singleton clusters around $r_{16}$ and $r_{17}$ are queued as shown in Figure 3.6. We then pull the first cluster from the queue, for example $C_7$, and determine its directly connected component which also contains $C_4$ and $C_5$. With $L_{G'}^{opt} = \{C_4, C_5''\}$, we add $C_5''$ back to $Q$. Next, we retrieve $C_8$ from the queue and run optimal clustering on its directly connected component which results in a new cluster $C_4'' = r_7 - r_9, r_{17}$. $C_4''$ is added again to $Q$. The next step is to retrieve $C_5$ which forms a directly connected component together with $C_4''$. When evaluated together, the optimal clustering outputs two new clusters, $C_4'$ and $C_5'$. Note that because $C_4''$ does not exist anymore, it has also been removed from the queue which now contains the two new clusters only. At this point, there are no remaining clusters in the queue and the algorithm finishes.

**Proposition 3.1.2** (Complexity of Iterative). The complexity of the Iterative algorithm is $O(|L_{G'}^{opt}| \cdot c(|G + \Delta G|))$. 

52
The complexity of Iterative depends on the optimal clustering before the increment $\mathcal{L}_G^{opt}$ as well as the incremental graph. As shown in the previous example, iterative causes overhead due to repetitive computation. Thus, its worst case scenario is not as good as for Connected. However, we expect and show that in practice there are many cases where Iterative terminates before exploring a substantial amount of the connected component and is thus more efficient. In terms of quality, we can show that Iterative is guaranteed to be optimal if the objective function satisfies all desirable properties.

**Theorem 3.1.6 (Optimality of Iterative).** Algorithm Iterative is optimal if and only if exchangeability, separability, and monotonicity hold.

*Proof.* We need to prove that if exchangeability and separability hold, Iterative obtains the same clustering on the monotone connected component as MonoConnected which is optimal under connectivity, locality, and monotonicity.

First, let us assume that exchangeability and separability hold. Let $G_1$ be the subgraph that contains all clusters in the result clustering but not in $\mathcal{L}_G^{opt}$. With $G_2$ as the subgraph with neighbor clusters of $G_1$ and $G_3$ be $\hat{T}(\Delta G) \setminus (G_1 \cup G_2)$, the iterative algorithm proceeds such that there has to be an optimal clustering of $G_1 \cup G_2$ which has no clusters across the border of $G_1$ and $G_2$. Given the exchangeability property, we observe that there has to be an optimal clustering for $G_2 \cup G_3$ with no cluster across the border of $G_2$ and $G_3$. In contrast, assume that there is a better clustering for $\hat{T}(\Delta G)$ which is referred to as $\mathcal{L}$. Due to separability, $\mathcal{L}$ has no cluster across all three subgraphs. Now let $\mathcal{L}_{12} \subseteq \mathcal{L}$ be the subset of clusters for nodes in $G_1 \cup G_2$ and $\mathcal{L}_3 = \mathcal{L} \setminus \mathcal{L}_{12}$ denote the cluster subset for $G_3$. In the next step, we replace $\mathcal{L}_{12}$ and $\mathcal{L}_3$ with those clusters obtained by applying Iterative. Because of the exchangeability property, the new clustering has to be optimal. Thus, $\mathcal{L}$ cannot be better than the clusterings obtained by applying Iterative.

If either exchangeability and separability do not hold, we can show that it is possible to construct an example in which the Iterative algorithm obtains a suboptimal clustering. First, consider exchangeability. If the exchangeability property does not hold, it is possible to update $G' \subset G$ in a way that violates the optimality of $\mathcal{L}_G^{opt}$ which in turn makes it unequal to MonoConnected. Second, if separability does not hold, then there always has to be a cluster spanning $G_1 - G_3$. However, if $G_1$ is the increment and $G_2 \cup G_3$ the original graph, then by the algorithmic nature of Iterative, it will always find a clustering that does not span all subgraphs which means that this property holds.
As a result, we can say that ITERATIVE is optimal for correlation clustering but not optimal for DB-index clustering.

3.1.1.3 Greedy Incremental Entity Resolution

The approaches that we have presented so far are optimal with respect to their corresponding batch objective function if certain properties hold. In practice, the optimal computation of most objective functions is intractable, i.e., it has been shown for example for correlation clustering, that solving the ER problem is NP-hard [BBC04]. We therefore next describe a greedy solution to incremental ER. It solves two problems. First, our greedy algorithm takes polynomial time thus achieving predictable, efficient performance for incremental ER. Second, it performs locally optimal operations that build upon previous clusterings. Using these we a) reduce the computational overhead by again only modifying those parts of the clustering that have actually changed and b) show experimentally that locally optimal operations at least approximate global objective functions.

Our greedy algorithm, GREEDY, uses the same methodology as ITERATIVE, i.e., it iteratively processes clusters and their directly connected component. We describe it in detail next. Afterwards, we show how it can be instantiated for specific clustering methods.

Algorithm

To greedily operate on clusters, we introduce three different operators, merge, split, and move. The idea behind the merge operator is to combine two clusters if their unified penalty is lower than the local penalty score for keeping them apart. Similarly, the split operator tests whether splitting an existing cluster into two clusters improves the penalty. Finally, the move operator examines whether records should be removed from one and added to another cluster to improve the score. We next describe all of these operators in detail and show how they can be combined to form our GREEDY algorithm.

Merge. The merge operation examines the relationship of two clusters. It decides whether to merge these clusters based on the local penalty score. For that, it computes two scores. One if the clusters are kept separate, one if they are merged. If the second is lower than the first, it will merge the clusters. Using same queuing mechanism as ITERATIVE, it will then put the newly merged cluster into that queue.

Formally, the merge operation follows these steps.
Table 3.1: Example for merge operation.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Dequeued</th>
<th>Enqueued</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>(C_7 = {r_{12}}, C_8 = {r_{13}})</td>
<td>({C_7, C_8})</td>
</tr>
<tr>
<td>2</td>
<td>(C_7)</td>
<td>(C''<em>2 = {r_5, r</em>{12}})</td>
<td>({C_8, C''_2})</td>
</tr>
<tr>
<td>3</td>
<td>(C_8)</td>
<td>(C'''<em>2 = {r_5, r</em>{12}, r_{13}})</td>
<td>({C'''_2})</td>
</tr>
<tr>
<td>4</td>
<td>(C''_2)</td>
<td>(C'<em>2 = {r_5, r_5, r</em>{12}, r_{13}})</td>
<td>({C'_2})</td>
</tr>
<tr>
<td>5</td>
<td>(C'_2)</td>
<td>-</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

1. For each neighboring cluster \(C'\) of \(C\), we determine the penalty score if they are merged, i.e., for \(C \cup C'\).

2. If a \(C''\) is discovered that meets this condition we a) merge \(C\) and \(C''\), b) add \(C \cup C''\) to the queue \(Q\), and c) remove \(C''\) from \(Q\) if previously \(C'' \in Q\) held.

Example 3.1.5 (Merge Operator). Going back to the running example, we now look at the integration of the two increments \(\Delta D_1\) and \(\Delta D_2\) with correlation clustering as objective function. To process \(\Delta D_1\), records \(r_{11}\) is put into a singleton cluster \(C_6\) first. As this record is not connected to any other cluster, there is no further operation that can be executed. For \(\Delta D_2\), we then generate two singleton clusters \(C_7 = \{r_{12}\}\) and \(C_8 = \{r_{13}\}\). They are added to \(Q\) as shown in Table 3.1. We then poll \(C_7\) from the queue and examine its related clusters. As the penalty of merging \(C_7\) with \(C_2\) is lower (0.1 vs. 0.9), they are merged into \(C''_2\). Similarly, enqueuing \(C_8\) and exploring its related clusters leads to a merge of \(C'''_2\) and \(C_8\). Finally, the resulting cluster \(C''''_2\) is related to cluster \(C_3\) containing \(r_6\). As the penalty score is lower if these two clusters merge (1.4 vs. 1.6), they thus form \(C'_2\).

Split. The second operator that we propose for the local decision-making of our greedy algorithm is the split operator. It is defined such that given a dequeued cluster \(C\), we examine whether splitting this cluster into at least two other clusters would be beneficial for the local penalty score. The operator proceeds in four steps.

1. For each of the nodes \(v \in C\), we determine whether the clustering \([C'', C\setminus C']\) with \(C' = \{v\}\) has a lower penalty score than \(C\).
Chapter 3: Data Integration Methodology

2. If we find a $v$ that matches the condition, we create $C'$ and continue. Otherwise, the operator execution finishes at this point.

3. For each of the other nodes $v' \in C \setminus C'$, we determine whether adding $v'$ to $C'$ improves the clustering score. If it does, we change $C'$ accordingly and repeat this step until all nodes have been examined for the latest $C'$.

4. Finally, $C$ and $C'$ are added to $Q$.

**Example 3.1.6 (Split Operator).** Picking up the running example, we now focus on increment $\Delta D_3$. Here, clusters $C_{11} = \{r_{14}\}$ and $C_{12} = \{r_{15}\}$ are added to $Q$ initially. Both of them are merged with $C_1$ which reduces the local penalty from 8.2 to 4 under correlation clustering. When we examine the new cluster $\{r_1 - r_4, r_{14}, r_{15}\}$, we find that splitting out $r_4$ further reduces the penalty to 2.2. At this point, there is no more node to be split and we terminate with two clusters $C'_1$ and $C''_1$.

**Move.** For each cluster $C$ in $Q$, the last operation that we check is whether the penalty decreases if we move a node from this cluster into another cluster $C'$ or vice versa. The corresponding algorithm proceeds as follows.

1. For each neighboring cluster $C'$ of $C$, i.e., for each cluster that $C$ is connected to, we check two node moving possibilities. First, we determine whether the penalty decreases if we move any node $v \in C$ to $C'$. Second, we determine whether moving a node $v' \in C'$ to $C$ results in a better clustering. If that is the case, create the new $C$ and $C'$ and repeat this step.

2. If $C$ and $C'$ have been modified, they need to be added to $Q$. At the same time the old $C'$ has to be removed from $Q$ if $C'' \in Q$ held.

**Example 3.1.7 (Move Operator).** Consider the $\Delta_4$ for this example. Inserting the records as well as applying the merge and split operators leads to the same clustering as shown in previously in Figure 3.6 after iteration 2. At this point, the clustering contains $C''_4$ with $\{r_7 - r_9, r_{17}\}$ and $C''_5$ with $\{r_{10}, r_{16}\}$. However, this clustering is not optimal. Instead, moving $r_9$ from $C''_4$ to $C''_5$ thus resulting in two new clusters $C'_4$ and $C'_5$ improves the penalty under correlation clustering from 2.4 to 2.2.
Algorithm 2. GREEDY($G(V, E), \Delta G, L_G$) algorithm.

**Input:** $G(V, E)$: Original similarity graph;  
$\Delta G$: Increment;  
$L_G$: Clustering of the original graph  

**Output:** Clustering $L_{G, \Delta G}$ as new $L_G$

1. $Q \leftarrow \emptyset$;  
2. $G' \leftarrow \overline{T}(\Delta G)$;  
3. Put each cluster in $G'$ into $Q$;  
4. while $Q \neq \emptyset$ do  
   5. dequeue $C \in Q$;  
   6. $\text{changed} \leftarrow \text{false}$;  
   7. // operations return true if they change the clustering  
   8. $\text{changed} \leftarrow \text{Merge}(C, G + \Delta G, L_G, Q)$;  
   9. if $\neg \text{changed}$ then  
      10. $\text{changed} \leftarrow \text{Split}(C, G + \Delta G, L_G, Q)$;  
   11. if $\neg \text{changed}$ then  
      12. $\text{changed} \leftarrow \text{Move}(C, G + \Delta G, L_G, Q)$;  
13. return $L_G$.

**Full Algorithm.** How the three operators are combined is shown in Algorithm 2. To initialize $Q$, we first compute the directly connected component surrounding each inserted, modified, or changed record. We then add the clusters that belong to these components to $Q$ (Line 3). Iteratively, we then poll clusters from the queue. For each of these clusters, we first examine whether the merge operation can be applied (Line 8). If that is not the case, i.e., if the clustering has not changed in the meantime, we proceed analogously with the split (Line 10) and move (Line 12) operator.

The operator execution order is determined by two factors. First, the move operator is more costly ($|C'||C''|$ comparisons) than the split operator ($|C|$ comparisons). Furthermore, the merge operator is cheaper than the split operator (2 comparisons). Second, we have experimentally verified that the most commonly applied operator is merge followed by split and move. This means that the merge operator causes the most changes to the clustering, pushing it to the end of the execution order would thus result in an increased execution time. Specifically, if a merge operation is applied, the new cluster needs to be checked again for all operators. Having frequent operations early on is thus preferable.
The iterative execution of these operators finishes when $Q$ is empty. At this point, $L_G$ has been adjusted such that $G$ now contains the increment $\Delta G$.

**Proposition 3.1.3 (Complexity of Greedy).** Let $g(|G|)$ be the time of evaluating the objective function on graph $G$. The complexity of the incremental Greedy entity resolution technique is then $O(|G + \Delta G|^6 g(|G + \Delta G|))$.

Greedy can be executed in polynomial time because evaluating the objective function can be done in polynomial time. Furthermore, although this bound seems high, we show that Greedy is efficient in practice for three reasons. First, the clusters are typically small, commonly significantly smaller than $G + \Delta G$. Second, we observe that the average number of related clusters per cluster is relatively small, i.e., much smaller than the worst case scenario of a connected singleton cluster for each record in $G + \Delta G$. Third, even though we have at most $O(|G + \Delta G|^3)$ clusters in $Q$, the number of clusters is much fewer in practice because increments commonly do not touch all clusters. Finally, we note that the approximation bound of the greedy algorithm remains an open problem. However, we show through an empirical study that it works well in practice.

**Instantiation for Correlation Clustering**

The instantiation of Greedy with correlation clustering as objective function can further simplify the greedy operators as we will show in the following for each operator. Note that the basic Greedy framework remains the same, we simply augment and optimize the execution.

**Merge.** Using the objective function of correlation clustering, Section 2.1.2.2, we determine that a merge is successful if for two clusters $C$ and $C'$ if the following equation holds for the new merged cluster:

$$\sum_{v \in C, v' \in C'} w(v, v') > \frac{|C| \cdot |C'|}{2}.$$ 

**Split.** Split can be simplified through two observations. First, we can order the records within a candidate cluster $C$ according to their connectivity within the cluster. Thus, we can immediately observe split candidates: If the connectivity drops below 0.5, we say that the record should be split. Second, if we have discovered a split node, we do not need to
consider each remaining record in \( C \) but instead the record \( v \) with the lowest difference of \( p_C(v) - p_{C'}(v) \) where \( p_C(v) \) is the sum of the edge weights for \( v \) and its connected records \( C \) and \( p_{C'}(v) \) is the sum of the edge weights of the connections between \( v \) and each record in \( C' \). If \( p_C(v) - p_{C'}(v) > \frac{|C| - |C'| - 1}{2} \) holds, we can stop the split operator execution. The difference can be maintained incrementally with augmentary data structures.

**Move.** The move operator can be modified similarly to the split operator. Specifically, instead of determining the relationship of all nodes \( v \in C \) with the reference cluster \( C' \) and vice versa, we incrementally maintain the connectivity of each \( v \) with its connected clusters. This allows us to determine the lowest \( p_C(v) - p_{C'}(v) \) and test whether this record can be moved. The move is successful if \( p_C(v) - p_{C'}(v) \leq \frac{|C| - |C'| - 1}{2} \) holds. The other direction is analogous.

**Proposition 3.1.4** (Complexity of GreedyCorr). *The complexity of the instantiation of Greedy for correlation clustering is \( O(|G + \Delta G|^6) \).*

Using the modifications proposed in this section, we reduce the execution time of the objective function to a constant which results in the above complexity for the with correlation clustering instantiated GreedyCorr.

**Instantiation for DB-Index Clustering**

If DB-index clustering is instantiated for greedy incremental execution (GreedyDB), we use the following simplifications. First, the separation measure does not need to be recomputed every time we evaluate the relationship of clusters. Instead, we store the its value and incrementally adept it whenever a merge, split, or move operation is initiated. Second, analogous to the ordering capacities of GreedyCorr, we can order our computation according to the separation measure. We observe that changes can only happen if the separation measure is modified significantly. That means that if the cluster modification candidates are listed in decreasing order of their computed separation measure change, we can prune the search space efficiently.

For DB-index clustering, we cannot prove the optimality of the merge, split, or move operator. However, we show empirically that this approximation works well in practice for DB-index clustering.
Chapter 3: Data Integration Methodology

Table 3.2: Statistics of real-world datasets according to Cautious.

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Biz</th>
<th>Cora</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>4892</td>
<td>1916</td>
</tr>
<tr>
<td>Avg #neighbors</td>
<td>3.05</td>
<td>41.8</td>
</tr>
<tr>
<td>Max #neighbors</td>
<td>26</td>
<td>106</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>2054</td>
<td>575</td>
</tr>
<tr>
<td>Avg #nodes</td>
<td>2.38</td>
<td>3.3</td>
</tr>
<tr>
<td>Avg #neighbors</td>
<td>0.7</td>
<td>27.8</td>
</tr>
<tr>
<td>Max #neighbors</td>
<td>10</td>
<td>92</td>
</tr>
<tr>
<td>Cluster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>1624</td>
<td>88</td>
</tr>
<tr>
<td>Avg #nodes</td>
<td>3.01</td>
<td>21.7</td>
</tr>
<tr>
<td>Max #nodes</td>
<td>29</td>
<td>18</td>
</tr>
</tbody>
</table>

3.1.1.4 Experimental Evaluation

In this section, we will show our experimental results on two real-world datasets and a synthetic dataset. They show that our incremental algorithms significantly improve over batch linkage on efficiency without sacrificing ER solution quality. At the same time, they significantly improve over naive incremental ER algorithms in terms of the result quality.

Experiment setup

Datasets. We use four different datasets for the evaluation and shortly explain them and their characteristics next.

Biz. The first dataset, Biz, contains 87 snapshots of business records in the San Francisco area. For the computation, we took the first snapshot as the original dataset, and computed an increment for each later snapshot. Every snapshot of this dataset contains approximately 5K records with slight variations depending on the applied update. The increments contain on average 120 Inserts, 118 Deletes, and 59 Changes, and the maximum number of operations in an increment is 4120. The top part of Figure 3.7(a) (with the Y-axis on the right side of the figure) shows a break down of the updates for each increment. We indexed the records on 3-grams for blocking. We then applied the Monge-Elkan [BM03] string similarity for pairwise similarity
computation and ignored edges with a similarity below 0.7. The similarity graph is relatively sparse as shown for the first snapshot of Biz in Table 3.2.

**Cora.** The second real-world dataset, Cora\(^1\) that we examine has been widely used for record linkage and contains 1916 publication records. On this dataset, we have a single snapshot, we indexed the records on words for blocking and, following [DN13], we applied the weighted Jaccard measure with a threshold of 0.9 for similarity computation. As Cora is not a naturally incremental dataset, we generated a range of possible increments as follows: In the first increment we randomly remove 1 record. In the \(i\)-th increment we add back the records removed in the \((i-1)\)-th increment and randomly remove \(2^{i-1}\) records. When we reach the last (i.e., the 11-th) increment, we only add back the previously removed 1024 records.

Comparing the two real-world datasets, we observe that the Cora dataset is more dense than the Biz dataset as shown in Table 3.2. This manifests in the higher number of average neighbors per node (41.8 vs. 3.05) and per cluster (27.8 vs. 0.7) as clustered by cautious correlation clustering. The impact of such different graph layouts on the effectiveness of our incremental techniques is discussed in detail later in this evaluation.

**Febrl.** Our first synthetic dataset uses the Febrl data generator.\(^2\) We vary the generation parameters which we will further explain in the part of our evaluation that focuses on the Febrl dataset specifically. For now it suffices to say that we can vary the expected number of records per cluster as well as their connectivity which in return varies the layout of the graph.

**Synth.** The second synthetic dataset that we use in this experimental evaluation is generated by creating perfect node clusters and then adding noisy edges to the dataset. It is used to explore the impact of connectivity on the execution time of our different approaches. Each generated graph is verified at least ten times and per parameter setting (further explained in the corresponding evaluation section), we generate at least ten different graphs.

**Implementation.** To determine the effectiveness of our incremental approaches, we implemented the following algorithms:

\(^1\)http://secondstring.sourceforge.net/

\(^2\)http://sourceforge.net/projects/febrl/
Chapter 3: Data Integration Methodology

**Batch** applies **Cautious** [BBC04] for correlation clustering and the hill climbing algorithm in [GDSZ10] for DB-index clustering.

**Naive** is the baseline incremental algorithm described at the beginning of Section 3.1.1. It compares each inserted record with existing clusters and either adds it into an existing cluster or creates a new cluster for it. Deleted records are removed from the clustering while changes to records are managed as a combination of the delete and insert operation.

**CC** applies **Connected** as described in Section 3.1.1.2.

**IT** applies **Iterative** as described in Section 3.1.1.2.

**Greedy** applies the **Greedy** algorithm with the corresponding initiation mechanisms as described in Section 3.1.1.3 for each clustering function.

Our implementation has two variations. Variation **CONT** is a continuously run incremental execution. That is, after each increment, the new clustering is the starting point for clustering the next increment. In contrast, the starting point in variation **RESET** is reset to the batch linkage results from the previous increment.

As mentioned above, the algorithmic implementation of **Batch** varies. First, we implemented the original approximation algorithm of [BBC04] for correlation clustering, also referred to as **Cautious**. Second, we implemented the **DB-Index** variation presented in [GDSZ10] as our reference objective function. Both algorithms require us to set algorithm-specific parameters. For **Cautious**, we use $\delta = 0.1$. In **DB-Index**, we set $\alpha = 0.2$ and $\beta = 0.1$. These parameters were chosen after experimenting with different values and choosing the best values in terms of their output quality. We implemented the algorithms in Java, and experimented on a Linux machine with eight Intel Xeon L5520 cores. The machine has 24GB RAM, i.e., the dataset as well as the current clustering and all auxiliary data structures are stored completely in memory.

**Measures.** We measure the performance and quality of our algorithms. To determine their performance, we repeated the experiments 100 times and report the average execution time. The reported time measurements are for the clustering time only. Pair-wise similarity computation as well as the time needed to perform blocking are orthogonal to the changes that we have made to the clustering computation. Thus, we will see (some sort of) improvement independent of these two performance enhancing techniques. For
quality, we report (1) the penalty (i.e., cut inter-cluster and missing intra-cluster edges) and (2) the *F-measure* if there exists a gold standard. Here, *precision* measures among the pairs of records that are clustered together, how many of these are correctly in the same cluster. Furthermore, *recall* measures among the pairs of records that refer to the same real-world entity, how many are correctly clustered together. Finally, the *F-measure* is computed as \( \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \).

**Objective.** The goal of our experiments is three-fold. First, we want to establish incremental record linkage as desirable in a dynamic environment because of performance improvement and quality consistency. Second, we will show that iterative incremental approaches can reduce the ER space drastically and decrease execution time even further. The last goal then is to identify the trade-offs between the three incremental algorithms introduced in this work.

**Experiments on Biz**

An overview of the quality and performance of our five implemented methods for the *Biz* dataset is given in Table 3.3. Correspondingly, Figure 3.7 shows the execution time of these algorithms per iteration under CONT on the left Y-axis. For the same setup, Figure 3.8 shows the penalty per iteration. We observe that all incremental linkage algorithms significantly improved over BATCH on performance. For correlation clustering, the slowest incremental algorithm reduced the execution time by 76.7% while the fastest one by 95.7%. As the implementation of correlation clustering is significantly more efficient than the DB-Index algorithm, we observe that for our second objective function, the slowest algorithm reduces the execution time by three orders of magnitude. At the same time, the fastest algorithm improves performance by nearly five orders of magnitude as BATCH takes 3.8 hours on average per snapshot. Among the incremental algorithms, the iterative algorithms (IT and GREEDY) outperform both NAIVE and CC in execution time and achieve comparable or better result quality.

**Observations for NAIVE.** The naive algorithm has competitive performance and output quality for small updates. However, as the performance of this algorithm is quadratic in the size of updates, it decreases with an increasing update size as shown in Figure 3.7 where large updates trigger execution times that are higher than those for BATCH (take as example iterations 23 and 37).
Table 3.3: Comparison of various algorithms on Biz. Highest performance is highlighted in bold. Penalty values are averaged. Improvement is calculated for Naive w.r.t. Batch and for the other methods w.r.t. Naive.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (s)</th>
<th>Impro.</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>3.7</td>
<td>-</td>
<td>988</td>
</tr>
<tr>
<td>Naive</td>
<td>0.86</td>
<td>76.7%</td>
<td>3037</td>
</tr>
<tr>
<td>CC</td>
<td>0.18</td>
<td>78.7%</td>
<td>988</td>
</tr>
<tr>
<td>IT</td>
<td>0.16</td>
<td>81.4%</td>
<td>981</td>
</tr>
<tr>
<td>Greedy</td>
<td>0.14</td>
<td>84.1%</td>
<td>592</td>
</tr>
<tr>
<td>Reset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>0.79</td>
<td>79.7%</td>
<td>1072</td>
</tr>
<tr>
<td>CC</td>
<td>0.20</td>
<td>74.2%</td>
<td>987</td>
</tr>
<tr>
<td>IT</td>
<td>0.17</td>
<td>77.7%</td>
<td>987</td>
</tr>
<tr>
<td>Greedy</td>
<td>0.20</td>
<td>74.3%</td>
<td>922</td>
</tr>
<tr>
<td>DB-Index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>997</td>
<td>99.9%</td>
<td>5426</td>
</tr>
<tr>
<td>CC</td>
<td>57.1</td>
<td>94.3%</td>
<td>651</td>
</tr>
<tr>
<td>IT</td>
<td>14.4</td>
<td>98.6%</td>
<td>783</td>
</tr>
<tr>
<td>Greedy</td>
<td>0.79</td>
<td>99.9%</td>
<td>941</td>
</tr>
</tbody>
</table>

Aggregated over all increments, Naive still improves execution time for correlation clustering by 76.7% and 99.9% for DB-index. Nevertheless, Naive is the slowest of our implemented incremental approaches. Naive is also the worst incremental approach in terms of quality which shows that a merge-only strategy does not take the changing graph structure into consideration appropriately. It has an average penalty of 3037 for correlation clustering and 5426 for DB-index which is three times, respectively six times, worse than any other incremental approach.

Observations for CC. In contrast to Naive, CC achieves basically the same result quality as Batch. Minor variations (for example an average penalty for Reset of 987 instead of 988) may occur as Cautious is an approximation and is thus not guaranteed to make the optimal decision. We furthermore observe that CC improves the execution time of Naive by at least 74.2% and that it is especially effective for large increments in correlation clustering. Table 3.4 shows an overview of the performance of CC in comparison to the other iterative approaches. A small update contains at most 50 updated records, a large update contains at least 500 updated records, all other updates are medium-sized. Our incremental dataset then consists of 51 small, 22 medium, and 13 large updates. The table shows how often either IT or Greedy is better compared to CC. We observe
3.1: Entity Resolution

(a) Correlation clustering on Biz.

(b) DB-index clustering on Biz.

Figure 3.7: Execution time comparison for CONT on Biz dataset.

(a) Correlation clustering on Biz.  (b) DB-index clustering on Biz.

Figure 3.8: Penalty comparison for CONT on Biz dataset.

that CC has worse performance than either GREEDY or IT in at least 92.2% (96.1% for DB-index) of the small and 86.4% (95.5% for DB-index) of the medium-sized updates. In contrast, CC outperforms IT in 46.1% of the large updates. More specifically, CC excels in those increments where the graph significantly changes, i.e., where a lot of nodes and edges are touched redundantly by the iterative approaches. An overview of which approach touches how many nodes and edges is shown in Table 3.5. The number of total nodes and edges describes the average number of nodes and edges that have been iterated over every increment where one node can be counted multiple times if, e.g., it is checked...
in both, a merge and split operation. As shown here, IT has lower values for both touched nodes and edges than CC which also explains the lower execution time. GREEDY touches more nodes and edges because it evaluates more clustering possibilities with its three operators. These operations are nevertheless efficient especially due to our selection criteria, especially for split and move. As a result, the execution time of Greedy is lower than CC for all DB-index experiments and at least better in 92.2% of the updates when correlation clustering is applied.

**Observations for IT and GREEDY.** Both, GREEDY and IT, have similar execution times for correlation clustering but for DB-index clustering GREEDY is 94.5% faster than IT. This difference can be explained by the applied objective function: Greedy does not use the hillclimbing algorithm but rather uses its three operators to determine a candidate clustering which reduces its execution time significantly. In general, we observe that IT outperforms GREEDY for small updates on correlation clustering where it is faster in 58.6% of the updates. Greedy inherently has an overhead for clusters that do not change because it checks for all three operations. In fact, we observe that on average Greedy attempts 115 merges, 44 splits, and 7 moves per iteration. Checking for all these operations is obviously more costly than IT if the modified cluster is not well-connected or will not change its clustering.
3.1: Entity Resolution

Table 3.5: Details on *Biz* (Correlation Clustering, CONT).

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>IT</th>
<th>Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>189</td>
<td>167</td>
<td>171</td>
</tr>
<tr>
<td>#Iterations</td>
<td>1</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>#Nodes</td>
<td>480</td>
<td>293</td>
<td>474</td>
</tr>
<tr>
<td>#Edges</td>
<td>2003</td>
<td>1482</td>
<td>1968</td>
</tr>
<tr>
<td>#Total-nodes</td>
<td>-</td>
<td>329</td>
<td>1237</td>
</tr>
<tr>
<td>#Total-edges</td>
<td>-</td>
<td>1817</td>
<td>11204</td>
</tr>
<tr>
<td>#Examined-Merge</td>
<td>-</td>
<td>-</td>
<td>115</td>
</tr>
<tr>
<td>#Real-Merge</td>
<td>-</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>#Examined-Split</td>
<td>-</td>
<td>-</td>
<td>44</td>
</tr>
<tr>
<td>#Real-Split</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>#Examined-Move</td>
<td>-</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>#Real-Move</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.6: Algorithm comparison for *Cora*. F-Measure is averaged.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (s)</th>
<th>F-Measure</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>5.24</td>
<td>0.811</td>
<td>10074</td>
</tr>
<tr>
<td>Cont</td>
<td>Naive</td>
<td>0.47</td>
<td>0.722</td>
</tr>
<tr>
<td></td>
<td>CC</td>
<td>3.3</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>IT</td>
<td>4.5</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>Greedy</td>
<td>1.09</td>
<td>0.837</td>
</tr>
<tr>
<td>Reset</td>
<td>Naive</td>
<td>0.318</td>
<td>0.754</td>
</tr>
<tr>
<td></td>
<td>CC</td>
<td>3.04</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>IT</td>
<td>4.31</td>
<td>0.811</td>
</tr>
<tr>
<td></td>
<td>Greedy</td>
<td>1.31</td>
<td>0.838</td>
</tr>
</tbody>
</table>

**Experiments on Cora**

Table 3.6 and Figure 3.9 show the execution time and quality of our proposed methods using correlation clustering on *Cora*. Here, similarity is computed using Jaccard metric. We observe that compared to *Biz*, the similarity graph for *Cora* is much denser, as explained previously. As a result, the experimental results and the effectiveness of our incremental...
Figure 3.9: Experimental results on Cora dataset.

The performance of the algorithms varies from our observations for the Biz dataset. We specifically show that for such a dataset, our Greedy approach provides good quality results as it performs local optimizations which are enhanced by the dense graph structure.

**Observations for Naive.** On small increments, Naive executes much faster than the other methods while having similar linkage quality. However, large updates also trigger a comparatively large number of mistakes made by Naive. The reason is the merge-only policy of Naive which does not allow to reconsider more beneficial clustering alternatives that may be possible if records are split or moved.

**Observations for Cautious algorithms.** Because of the high connectivity of this
3.1: Entity Resolution

dataset, CC finished in nearly the same time as BATCH starting from the fifth increment. Indeed, in this round CC touches 64% of the nodes in the graph. Similarly, the runtime of IT is even higher than the runtime of CC after the fifth increment. The reason is that IT examines nearly the same subgraph as CC.

**Observations for Greedy.** The Greedy algorithm is much faster than IT on this dataset and reduced the execution time by 66.8% over CC and by 75.8% over IT. Recall that we are using Cautious as implementation for both CC and IT. This algorithm decides whether to combine records into a cluster based on set logic which obviously gets more expensive the more connected the graph is because more candidates need evaluation. In contrast, merge operations as suggested for Greedy are structured much more simply: The penalty of two clusters separated or combined is computed following the objective function and we then choose the better solution. An efficient merge operator thus decreases execution time effectively, making up for the more time-costly split and move operations.

**Experiments on Febrl**

As the Febrl dataset is synthetically generated, we first explain the different parameter choices for this dataset after which we examine its performance and result quality.

**Data generation.** We generate synthetic data using the Febrl dataset generator. It allows us to specify how many original and duplicate records we generate as well as the distribution of noise within the generated dataset. For our experiments we vary the following parameters: (1) The number of duplicates per original record \(d\), (2) the number of modifications within one attribute and how many attributes are modified \(m\) (we use the same value for both as we want to show the behavior of our approaches if the noise level increases, separating them is less relevant), (3) the number of inserted and deleted nodes within one increment \(n_{ins}\) and \(n_{del}\), and (4) \(\theta\), the similarity threshold applied in generating the similarity graph. For comparability, we generate 10K original records and 10k duplicate records as the original dataset given \(d\) and \(m\). Note that \(\theta\) modifies the internal similarity graph but not the underlying dataset. We use the blocking generated by Febrl, which guarantees that duplicates are in the same block. To determine the pairwise similarity between records, we use Jaro Winkler similarity, as suggested in [DN13] for Febrl data. We apply correlation clustering as the objective function in all reported experiments. Like Cora, Febrl is not a naturally incremental dataset. As a result, we need to simulate the increments. Every experiment that we run has 10 increments, by default we randomly
select 10K records as the initial dataset. We then randomly selected $n_{ins} = 1K$ (i.e., 5%) records from the remaining records for insertion and $n_{del} = .5K$ (i.e., 2.5%) existing records for deletion per increment. We use $d = 9$, $m = 3$, and $\theta = .8$ as default values for the remaining parameters.

**Results:** We make four observations when varying these parameters: First, varying $\theta$ confirms the observations made for Biz and Cora. If the graph is denser, i.e., the average number of neighbors increases from 0 ($\theta = 1$) to 25.01 for ($\theta = .7$), Greedy is more resilient in terms of quality and requires less execution time than the Cautious ap-
proaches (IT and CC) for denser environments (Figure 3.10(a) and Figure 3.10(b)). We also observe (not shown in the graph) that GREEDY improves the result quality with each increment because it touches more nodes. Second, increasing \( d \) from 1 to 9 monotonically increases the average execution time by a factor of 1.1 (NAIVE), 1.7 (CC), 4.6 (IT), and 1.4 (GREEDY). The quality for all approaches increases from \( d = 1 \) at 0.61 (NAIVE), 0.59 (IT and CC), and 0.68 (GREEDY) to \( d = 9 \) with 0.82 (NAIVE), 0.87 (IT and CC), and 0.9 (GREEDY). The increase can be attributed to the fact that more duplicates mean a better chance of having high cohesion within an entity. Third, an increase of \( m \) results in a similar performance per method as shown in Figure 3.10(a) for \( \theta = .8 \). At the same time, the change causes a decrease in F-measure for all approaches. More specifically, the quality of NAIVE, CC, and IT decreases by 5\% increasing \( m \) from 1 to 5 while the result quality of GREEDY is decreased by 1\%. Finally, we note that delete-only workloads are more efficient to process for all approaches except for CC. It takes the whole connected component as input which makes it indifferent to workload characteristics while IT, GREEDY, and NAIVE a) evaluate where the inserted or deleted record fits in and then b) how the clustering changes because of that. For deleted records, the first part is obviously less costly as the position of a record in the clustering is already known. Specifically, we observe an increase by factor 60 for NAIVE, 8 for IT, and 2 for GREEDY when comparing a delete-only to an insert-only workload (\( n_{\text{ins}} = 1K \) and \( n_{\text{del}} = 1K \)). The varying factors are directly correlated to the way that deletes are handled for each approach: While GREEDY creates the clustering with costly split and move operations, IT initiates a cluster check on a smaller, less connected cluster, and NAIVE simply removes the record from the dataset. If we focus on an insert-only workload, we can show how the algorithms scale differently in noisy environments (Figure 3.10(c)): While the performance of NAIVE is 52\% better than GREEDY for \( n_{\text{ins}} = 100 \), GREEDY is 47\% better than NAIVE for \( n_{\text{ins}} = 1000 \). At the same time, GREEDY maintains a higher F-measure (.9 compared to .82 of NAIVE). We can observe in this figure that a high connectivity of the graph is highly correlated with the performance of CC and IT: As both approaches (iteratively) explore at least 64.4\% of the records in the graph for \( n_{\text{ins}} = 1000 \), their performance is clearly worse than GREEDY which only evaluates 13\% of the graph on average. These observations are confirmed by varying the threshold \( \theta \) thus increasing the level of noise as shown in Figure 3.10(a).
Figure 3.11: Execution time for varying synthetic datasets.

Experiments on Synth

For these experiments, we modify a snapshot of synthetic data by adding noisy edges to a perfect graph. These edges are generated with a) a uniform error distribution and b) a zipf skewed distribution. We furthermore vary the amount of noise inserted into the graph and show the performance of our approaches in 3.11 for correlation clustering. First, we observe that the performance improvement of the incremental approaches compared to the original batch algorithm is several orders of magnitude if the snapshot remains unchanged. Then, we note that the amount of noise in the synthetic graph determines the execution time of the incremental approaches. That is, the lower the number of noisy edges, the better the performance of the incremental approaches. This observation is directly correlated to the core idea of our techniques: We limit the search space efficiently by minimizing the overhead of recomputing parts of the graph that remain unchanged. However, if many noisy edges are part of the update and thus a large part of the graph is (in-)directly affected by the update’s changes, the incremental approaches become less efficient. In this set of experiments, we observe for higher connectivity of the graph, i.e., more noisy edges in the graph, that both CC and IT have a slightly lower performance than the original batch algorithm on average. This is a result of the additional overhead of computing the connected component (CC) resp. the additional iterations the algorithm has to execute (IT). However, we also observe that our locally optimal approximation, Greedy, performs
better than either of the other incremental approaches as well as the batch algorithm. The good performance of GREEDY is due to its fast local decision-making which allows it to converge faster than either of the approaches with a global objective function. In contrast to CC and IT, GREEDY can thus update a larger portion of the graph in a manner which makes it even more efficient than the original batch technique.

Finally, comparing the differences in the predictability of the execution time of the two different synthetic data distributions, we observe that the skewed distribution produces graphs where the performance of the algorithms is more stable with different update sizes. This follows naturally from the character of these algorithms: Given a random assignment of noisy edges, the (indirectly) updated part graph may vary drastically which stabilizes the larger the update gets. However, with a zipf-based distribution, the graph composition is statistically more similar between different graph generations. Given a real-world dataset, we expect the error model to be a mixture of these two distributions. Specifically, many errors such as spelling or formatting mistakes lead to noise within a limited part of the graph. However, not all nodes in the graph commonly are related which means that there are potentially many smaller sources for the noise distributed throughout the graph. This explains why CC and IT perform comparatively better in real-world settings than in these synthetic experiments.
3.1.2 Fault-Tolerant Entity Resolution with the Crowd

In this part of our work, made publicly available in [GNK+15] and [GKRW12], we discuss how crowdsourcing modifies the problem of entity resolution. Several studies have recently shown that crowdsourcing can produce higher quality solutions for a subset of data integration tasks [GWKP11, WLK+13]. For example for complex problems such as entity resolution (ER) or picture classification, crowdsourcing has been established as an alternative to automated techniques. In fact, approaches that prune the search space with automated ER mechanisms and then enhance data quality through crowdsourcing are common for a large number of high profile ER systems such as the Google Knowledge Graph [Sin12] or the Facebook Entities Graph [Fac16]. Even though the overall result quality of ER solutions generally benefits from human input, it can also be observed that crowd workers may make mistakes when executing tasks. These mistakes may be the result of carelessness, ambiguities in the task description, or even malicious behavior. More specifically, it is common to have crowd error rates as high as 30% [IPW10] on well-established crowdsourcing platforms. Figure 3.12 shows an example for a task which may mislead humans and how these mistakes could be avoided intuitively.

Example 3.1.8 (Animal Classification). Crowd workers are given the task to determine which animals belong to the same breed, i.e., \( r_1 \) and \( r_2 \) are seals, \( r_3 \) and \( r_4 \) show sea lions. However, we observe that distinguishing the full-grown animals (records \( r_2 \) and \( r_4 \)) is easier than telling the baby animals (records \( r_1 \) and \( r_3 \)) apart because they have similar appearance characteristics.

In our work, we study the problem of crowdsourced entity resolution with potentially erroneous input by crowd workers which we refer to as fault-tolerant entity resolution. We choose fault-tolerant as naming convention because crowd workers make faulty decisions that have to be tolerated by the ER engine. The terminology here indicates that we take all available information from the crowd into consideration when making robust decisions about the ER solution.

As worker input is usually not for free, our goal is to minimize the monetary cost while maximizing the overall ER result quality. Our solution consists of two components: First, we address data interpretation, i.e., how the answers provided by crowd workers lead to an ER solution, and discuss how it can be efficiently implemented for ER computation with potentially erroneous crowd answers. Second, we focus on minimizing the cost that crowdsourcing incurs which is also called the next-crowdsour problem.
3.1: Entity Resolution

Data Interpretation. Prior work in crowdsourced entity resolution [VBD14, WLK+13, WLGM13] has focused on handling data interpretation and crowd worker quality as separate problems. For instance, this line of work proposed to use qualification tasks [KKMF11] to filter out malicious or low-performing workers or to use replicated tasks with quorum votes (e.g., majority of answers) to determine the correct answers in ambiguous cases [NR10]. If data quality is thus ensured, the ER algorithm can be designed under the assumption that there are no conflicts in the dataset. In contrast, we argue in our work that there are numerous sources for erroneous information from the crowd that a good ER algorithm needs to interpret correctly. Consider the following example: If workers misclassify the baby sea lion and seal (r1 and r3) with three positive and two negative votes and correctly identify the sea lion (r1 and r2) and seal pairs (r3 and r4) then all of these animals would be wrongly classified as the same species. Under the assumption of a majority-based decision scheme, even close decisions such as [r1, r3] are not questioned although the indecision of the crowd clearly indicates that this relationship is uncertain and should be explored further.

To overcome these mistakes, we propose a graph-based path ER technique to identify and appropriately interpret noisy data. Our technique considers both positive and negative (indirect) crowd answers between two records and provides provably better quality than majority-based approaches that only value the dominant decision. In addition, we discuss how to integrate available worker input efficiently into the decision making process, allowing for a consistent ER solution at any point in time.
Next-Crowdsource Problem. In addition to interpreting noisy data, we look at ways to minimize the cost that ER with erroneous crowd information incurs. In that context, we focus on (a) crowd task ordering and (b) task parallelization strategies. Generally, crowdsourced ER is executed on platforms such as Amazon Mechanical Turk [Ama16] which allow task requesters to employ crowd workers for monetary compensation. These task providers therefore need to devise effective task assignment strategies that minimize the overall monetary cost by maximizing the information gain per cost unit. To address the challenge of task ordering, we develop three different ordering strategies that can be employed in the context of crowdsourced ER and examine them with synthetic and real-world datasets. The second technique that we explore in this part of our work is task parallelization. It is a mechanism that tries to minimize the end-to-end runtime of crowdsourced ER by publishing multiple tasks at the same time on the platform. In our discussion of parallelization strategies, we show the trade-off between the runtime acceleration of crowdsourced ER and its monetary cost and output quality.

In summary, we make the following contributions:

Fault-tolerant entity resolution. We formally introduce fault-tolerant decision functions for ER that decide whether two records belong to the same entity or not. To handle unreliable information, we devise a path-based graph interpretation mechanism and define a clustering algorithm that computes the ER solution based on noisy pair-wise decisions to address the data interpretation problem.

Cost and quality optimization. Given that the crowd is expensive to employ on a large scale, we discuss task ordering and parallelization strategies and the impact of these mechanisms on data quality and incurred cost. These mechanisms are essential to solve the next-crowdsourcer problem.

3.1.2.1 Problem Overview

In this work, we discuss mechanisms to enable entity resolution with imperfect answers from crowd workers. In order to accurately capture the information provided by them, we need an efficient data structure that allows us to encode both their positive and negative signals. For this purpose, we introduce the notion of a votes graph that stores this information and will later allow us to efficiently interpret the crowd worker’s answers. An example for a votes graph can be seen in Figure 3.12.
Definition 3.1.10 (Votes Graph). A votes graph \( G = (R, E) \) is a weighted undirected graph that consists of records \( R \) as its nodes and a set of edges \( E \) which determine the direct relationship between any records \( r_i \) and \( r_j \), with \( r_i, r_j \in R \). For each record pair \( [r_i, r_j] \) there exists at most one positive edge \( p_{ij} \) and one negative edge \( n_{ij} \). They correspond to the positive and negative crowd answers for this record pair. That is, while \( p_{ij} \) corresponds to the total number of votes that say that \( r_i \) and \( r_j \) belong to the same real-world entity, \( n_{ij} \) are the votes against it.

For simplicity, we assume that each vote from each data source (i.e., a specific worker) has the same weight. However, note that weighting schemes for (un-)reliable crowd workers can be easily integrated into the votes graph: For example if the system assumes a crowd worker to have complete knowledge, it can transform that worker’s votes to the maximal edge weight. Other transformations can be computed analogously.

ER Framework. There are two steps that are integral to automated ER systems on a (votes) graph \( G \). First, all pair-wise similarities for any two records \( r_i \) and \( r_j \) are computed. Second, a clustering \( C \) is computed that optimizes the record-to-entity assignment according to some objective function, for example those previously introduced in Section 3.1.1. This well-established pipeline for automated ER is similar but not equivalent to the pipeline for crowdsourced ER. Do understand the core differences, remember that the comparison of records in a crowdsourced environment raises two questions:

1. Given that each additional edge incurs a monetary cost, which edges do I really need to know about?

2. And if \( G \) is incomplete and there exist no direct votes between \( r_i \) and \( r_j \), how can we estimate the similarity between them?

The first question is raised due to budgetary limitations that are inherent to crowdsourcing applications. Through these limitations and the choice of observed edges, other record pairs may only be known indirectly or not at all. To predict an accurate record-to-entity assignment, it is therefore necessary to estimate their relationship. ER solutions that are designed for the crowd therefore focus on two slightly different core problems than traditional ER: The first problem is how to understand, interpret, and enrich the data that has been retrieved from the crowd which we refer to in the following as the data interpretation problem. The second problem is to determine the next-crowdsource order in which record pair information is requested from the crowd.
We have previously explained basic principles of crowdsourced ER (refer to Section 2.2.3 for details) and shown a generic algorithm to solving the problem. As discussed, there are three categories that current research in crowdsourced ER can be separated into: (a) fault-tolerant exhaustive, (b) fault-tolerant, and (c) consensus-based ER strategies. In our work, we address the first two categories. That is, we take all provided information by the crowd when solving the data interpretation and next-crowdsource problems.

Problem Definition. To formalize the these two problems, we first define what an optimal solution entails in crowdsourced ER. For the data interpretation problem, the goal is to find a clustering $C^*$ that represents the correct entity resolution solution, i.e., the record-to-entity mapping is equivalent to some ground truth. Finding such a clustering is straightforward if all pair-wise similarity estimates are correct. In other words, if there exists an oracle that knows whether any two records $r_i, r_j \in R$ belong to the same real-world entity then finding $C^*$ with any of the well-established ER algorithms is trivial. The reason is that there are no contradictions in the edge set (i.e., there are no $i, j, k$ s.t. $r_i = r_j$ and $r_i = r_k$ but $r_j \neq r_k$). That means that there exists one unique clustering $C^*$ that is consistent with the specified pair-wise relations.

Under the assumption of incomplete or incorrect edges, i.e., erroneous votes from the crowd workers, finding $C^*$ becomes an optimization problem. Specifically, we want to find the clustering $C$ that is either equivalent to $C^*$ or the best approximation thereof based on the current state of the votes graph $G$. To compute $C$, our algorithms estimate the distance to the optimal solution through a distance measure $d$ which represents the similarity of $C$ to $C^*$. It is essentially an objective function that estimates the correctness of a solution based on the clustering mechanics that define $C^*$. Examples for $d$ include minimizing the number of negative similarity scores within the same and positive scores across clusters, or to maximize the number of positive edges within a cluster. The choice of optimization metric is dependent on the applied ER algorithm. However, assume in the following that the goal for all candidate mechanisms is to minimize $d(C, C^*)$ where $d(C, C^*)$ is the distance of $C$ to $C^*$ for simplicity. We can then formulate the data interpretation problem as follows.

Problem 3.1.1 (Data Interpretation Problem). Given are all record pairs $[r_i, r_j] \in G$ and a metric $d(C, C^*)$ that assigns a distance for any given clustering $C$ from ground truth $C^*$. The data interpretation problem is to find the best clustering $C$ such that $d(C, C^*) < d(C', C^*)$ for any alternative clustering $C'$ of $G$. 

78
3.1: Entity Resolution

The second problem of ER with a crowd is that crowdsourcing platforms incur cost for the task requester. The goal when issuing tasks, i.e., obtaining more information for specified edges in \( G \), therefore becomes finding those tasks that provide maximal information for minimal cost. Minimizing the task space through intelligent vote requests is part of the next-crowdsource problem. In that context, task ordering is essential for efficient crowdsourced ER because it is highly correlated to the output quality in addition to the required (monetary) cost. To understand why, remember that the system needs to estimate pairwise decisions in the data interpretation problem. Thus, it provides better solutions if it knows which questions enable fast convergence to the best possible clustering solution. The next-crowdsource problem can therefore be formalized as follows.

**Problem 3.1.2** (Next-Crowdsource Problem). *Given a votes graph \( G \) and a distance function \( d(C, C^*) \). The next-crowdsource problem is to choose a record pair \([r_i, r_j]\) \( \in G \) for a crowd worker to vote on, such that adding the outcome of the vote to \( G \) minimizes the expected distance \( d(C', C^*) \) of the updated clustering \( C' \).*

**Worker Quality.** The quality of the entity resolution result is based on the quality of the answers the crowd workers provide. Formally, we define a correct answer for a record pair \([r_i, r_j]\) as the true positive or true negative answers, i.e., if \( r_i \) and \( r_j \) belong to the same entity, a correct answer is ‘Yes’. Similarly, a faulty answer for \([r_i, r_j]\) encompasses false positive and false negative answers of crowd workers. In our experimental evaluation, we show that faulty crowd answers have drastic impact on the ER quality as observed in pair-wise precision and recall (see Section 3.1.2.5 for details). Incorrect answers are often the result of incomplete knowledge in a certain domain or lack of attention during task execution. In fact, we observe an evenly distributed error rate for both false positive and false negative crowd answers across all workers. For example, we employed 545 different workers to examine our landmarks dataset (see Section 3.1.2.5 for details) out of which only 19 workers (i.e., 3.5% of all workers) deviated significantly from the measured average worker quality because of errors in their answers. Out of these, only 9 did more than 20 tasks and thus had more significant impact on the results.

The algorithms presented in this work do not differentiate between crowd workers but rather handle every worker equally. The reasons for that are two-fold. First, as mentioned above, we observe little variation in worker quality in our real-world experiments. Second, the same crowd is often not available for the same task more than once. As a result, available workers may not have a quality profile which the algorithm can fall back on. For
example, in our landmarks dataset, we observed that only 51.3% of the workers executed more than 20 tasks. Obviously, it is only possible to build accurate error statistics per worker if these workers provide sufficient sample work. To compensate for missing error profiles, the general decision strategies that we explore in this work employ robust decision mechanisms that make them independent of worker error statistics.

### 3.1.2.2 Data Interpretation Problem

A decision function is a function that determines the relationship of \( r_i \) and \( r_j \) based on \( G \). To correctly interpret the potentially noisy votes in \( G \), it needs to fulfill several properties which we define next. We then introduce a novel decision function called MinMax that provides fault-tolerant data interpretation and adheres to these properties. To contrast MinMax with alternative pair-wise decision functions, we discuss its (dis-)advantages in Section 3.1.2.2.

#### Properties of Pair-Wise Decision Functions

Given a votes graph \( G = (R, E) \), a desirable pair-wise decision function forms a decision about the relationship of two records \( r_i, r_j \in R \) by evaluating the information contained in the edge set \( E \) of the votes graph.

**Definition 3.1.11** (Decision Function). A decision function \( f \) evaluates the relationship of two records \( r_i \) and \( r_j \in R \) by first finding all distinct acyclic paths \( H = (r_i, \ldots, r_j) \) connecting \( r_i \) and \( r_j \). It forms its decision based on the \( \mathit{p}_{kl} \) and \( \mathit{n}_{kl} \) votes that are part of these paths, for each \([r_k, r_l] \in H\). The result of \( f(r_i, r_j) \) is then either of three decision ‘yes’, ‘no’, or ‘unknown’, which describes whether \( r_i \) and \( r_j \) belong to the same entity.

Any decision function should obey all mathematical properties of an equivalence relation such as the ‘same-entity-as’ relation. More formally, we expect the following properties from such a decision function \( f \):

**Reflexivity.** For any record \( r_i \) and any votes graph:

\[
f(r_i, r_i) = \text{'yes'}
\]

**Symmetry.** For records \( r_i \) and \( r_j \) and any votes graph:

\[
f(r_i, r_j) = f(r_j, r_i)
\]
Consistency. If the decision function decides that two records \( r_i \) and \( r_j \) point to the same entity, then there exists positive information between \( r_i \) and \( r_j \). Likewise, there has to exist negative information for a ‘no’ decision.

Convergence. For every connected record pair \([r_i, r_j]\), there has to exist an acyclic path \( H = (r_i, \ldots, r_j) \) connecting \( r_i \) and \( r_j \) which is computable. If \( r_i \) and \( r_j \) are unconnected, the default decision of the function is ‘unknown’.

Transitivity. For the three records \( r_i, r_k, \) and \( r_j \) and any votes graph:
\[
f(r_i, r_k) = \text{‘yes’} \land f(r_k, r_j) = \text{‘yes’} \implies f(r_i, r_j) = \text{‘yes’}
\]

Anti-transitivity. For the three records \( r_i, r_k, \) and \( r_j \) and any votes graph:
\[
f(r_i, r_k) = \text{‘yes’} \land f(r_k, r_j) = \text{‘no’} \implies f(r_i, r_j) = \text{‘no’}
\]
\[
f(r_i, r_k) = \text{‘no’} \land f(r_k, r_j) = \text{‘yes’} \implies f(r_i, r_j) = \text{‘no’}
\]
Consistency guarantees that the decision function forms appropriate decisions. Convergence of a decision is furthermore required because it guarantees that the decision function will always make a decision. Specifically, even if there exist cycles in the votes graph, the acyclic path generation will always generate a path between records \( r_i \) and \( r_j \). (Anti-)transitivity is an essential tool for cost-conscious environments such as ER in a crowdsourcing setup. Decision functions apply it because in contrast to traditional ER, each information request incurs additional monetary cost which is to be avoided. As a result, both transitivity and anti-transitivity are core concepts of decision functions in state-of-the-art ER solutions [WLK+13].

MinMax Similarity Measure

The MINMAX pair-wise decision function \( f_M \) uses both positive and negative information in the crowd workers’ input and establishes a similarity measurement for every record pair through path-based inference. It is a novel fault-tolerant technique that is inspired by work on preference functions [FW00] and voting schemes that discuss ranking pair-wise decisions [Sch11]. While these mechanisms are used for decision-making in a space where crowd signals are one-dimensional (i.e., \( r_i \) and \( r_j \) belong together or \( r_i \) is better than \( r_j \)), obtaining information from the crowd for ER problems enables both positive and negative decision signals (i.e., \( r_i \) and \( r_j \) belong or do not belong together). MINMAX uses the notion of positive and negative acyclic paths in the votes graph to evaluate whether two
records belong to the same entity. The score of a positive path between records \( r_i \) and \( r_j \) is denoted as \( p^*_{ij} \) while the negative path scores are referred to as \( n^*_{ij} \).

**Definition 3.1.12 (MinMax Decision Function).** Given two records \( r_i \) and \( r_j \) decides whether \( r_i \) and \( r_j \) belong to the same entity given the positive votes \( p^*_{ij} \) and the negative votes \( n^*_{ij} \) along the path(s) connecting these two records.

\[
f_M(r_i, r_j) = \begin{cases} 
  \text{Yes}, & \text{if } p^*_{ij} - n^*_{ij} \geq q_p \\
  \text{No}, & \text{if } n^*_{ij} - p^*_{ij} \geq q_n \\
  \text{Do-not-know,} & \text{otherwise}
\end{cases}
\]

MinMax decides that \( r_i \) and \( r_j \) belong to the same entity if there is sufficient evidence for such a decision represented through a quorum \( q_p \) (\( q_n \) for negative decisions). The higher \( q_p \) and \( q_n \), the more crowd workers need to support a positive (negative) decision which ensures better result quality. In practice, we show that a quorum as low as 3 is sufficient for accurate decision making as shown in Section 3.1.2.5.

**Definition 3.1.13 (Path Definition).** A positive path in a votes graph is a sequence of records \( H_p = (r_i, \ldots, r_j) \) connecting records \( r_i \) and \( r_j \) such that all consecutive record pairs \([r_k, r_l] \in H\) have only positive weights, i.e., \( p_{kl} > 0 \). A negative path \( H_n = (r_i, \ldots, r_j) \) contains exactly one negative record pair \([r_k, r_l] \) such that \( n_{kl} > 0 \). All other record pairs on \( H \setminus [r_k, r_l] \) have positive weights with \( p_{kl} > 0 \).

The notion of paths allows for effective transitivity: If two records \( r_i \) and \( r_j \) are connected through a positive path with high weights and no negative paths, it is likely that they belong to the same entity. Anti-transitivity on the other hand can be leveraged with exactly one negative edge in the path only: If \( r_i \neq r_k \) and \( r_k \neq r_j \), there is no way to automatically infer the relationship of \([r_i, r_j]\).

**Example 3.1.9 (Paths).** Going back to the introductory example (Example 3.1.8), imagine that we want to determine the positive and negative paths connecting \( r_2 \) and \( r_3 \). There exist negative paths \( H_{n_1} = (r_2, r_4, r_3) \) and \( H_{n_2} = (r_2, r_1, r_3) \). There also exists exactly one positive path \( H_p = (r_2, r_1, r_3) \) that has only positive path scores.

The core idea of the MinMax decision mechanism is that a path is only as strong as its weakest link, i.e., the edge in the path that has the lowest weight. Both positive and negative paths are thus assigned a score according to the minimum weight of all absolute
3.1: Entity Resolution

(a) Internal representation (Example 3.1.10).

![Votes Graph]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>-</td>
<td>+3</td>
<td>+1</td>
</tr>
<tr>
<td>r₂</td>
<td>-1</td>
<td>-</td>
<td>+1</td>
</tr>
<tr>
<td>r₃</td>
<td>-3</td>
<td>-3</td>
<td>-</td>
</tr>
<tr>
<td>r₄</td>
<td>-3</td>
<td>-4</td>
<td>-</td>
</tr>
</tbody>
</table>

![MinMax Matrix]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>r₂</td>
<td>-3</td>
<td>-</td>
<td>+3</td>
</tr>
<tr>
<td>r₃</td>
<td>-3</td>
<td>-3</td>
<td>-</td>
</tr>
<tr>
<td>r₄</td>
<td>-3</td>
<td>-3</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Noise detection (Example 3.1.12).

Figure 3.13: MINMAX computation examples.

edge weights. If there exist multiple paths that connect two records, we choose the path with the maximal score as it signifies a higher confidence from the crowd. We call the path scores for a record pair \([r_i, r_j]\) the positive score \(p^*_{ij}\) resp. the negative score \(n^*_{ij}\) of a path.

**Definition 3.1.14 (Path Scores).** Given records \(r_i\) and \(r_j\) ∈ \(R\), let \(H_{ij}^p\) denote the set of positive paths connecting \(r_i\) and \(r_j\). For every path \(H_\mu\) ∈ \(H_{ij}^p\), its score \(h(H_\mu)\) is computed as the minimum of all direct scores for any consecutive record pair \([r_k, r_l]\), i.e., \(h(H_\mu) = \min(p_{kl})\), for all \([r_k, r_l]\) ∈ \(H\). The MINMAX path score \(p^*_{ij}\) is then computed as \(\max(h(H_\mu) \forall H_\mu \in H_{ij}^p)\). Negative path scores \(n^*_{ij}\) are computed analogously.

Positive and negative MINMAX scores can be represented as a two-dimensional matrix of record pairs: Positive path scores correspond to the upper triangular part of the matrix while negative paths to the lower triangular. If there exists no positive or negative path between two records, \(p^*_{ij}\) resp. \(n^*_{ij}\) is set to 0.

**Example 3.1.10 (MINMAX Computation).** To see how the MINMAX scores are computed, take a look at Figure 3.13(a) where green edges signify positive and red edges equal negative
Chapter 3: Data Integration Methodology

votes. To make a decision for \([r_3, r_4]\), two positive paths are examined: \((r_3, r_4)\) with a score of 3 and \((r_3, r_1, r_2, r_4)\) with \(\min(1, 3, 1) = 1\). The score for \(p_{(3,4)}^*\) is then computed as \(\max(3, 1) = 3\). To compute the negative score \(n_{(3,4)}^*\), we observe that there exists only one negative path, \((r_3, r_1, r_2, r_4)\). The negative path score \(n_{(3,4)}^*\) is thus \(\min(1, 3, 4) = 1\).

Computing both, negative and positive, paths ensures fault-tolerant data interpretation. At the same time, it also gives the MINMAX decision mechanism the power to adapt to inconsistencies found in crowd responses: Low confidence decisions can be outweighed through stronger evidence for the opposite decision.

**Example 3.1.11 (Fault-Tolerance).** Imagine that in Figure 3.13(a), the edges between \(r_2\) and \(r_4\) do not yet exist. The positive path connecting all records then indicates that they all belong to the same entity albeit with low confidence, i.e., \(p_{24}^* = \min(3, 1, 3) = 1\). Introducing the new edge evidence between \(r_2\) and \(r_4\) weakens the previously made decision and causes the entity to be split into two separate entities. The negative edge between \(r_2\) and \(r_4\) now propagates to \(r_1\) \((n_{14}^* = \min(4, 3) = 3)\) and \(r_3\) \((n_{23}^* = \min(4, 3) = 3)\) respectively.

Choosing the weakest link of a path provides a mechanism to cautiously evaluate a path. Nevertheless, using absolute values has one general drawback: If a decision is based on a positive score outweighing a negative score or vice versa by a specified margin, dominating values can cause incorrect inference as a result of incomplete information. Thus, an algorithm relying on such information may make a decision for \([r_i, r_j]\) for which intuitively a decision should not be made. Imagine two records that are connected through a positive edge \((r_k, r_l)\) which dominates the path score, i.e., it is the minimal edge in the path. We say that \((r_k, r_l)\) is dominated if its counterpart, here the corresponding negative edge between \(r_k\) and \(r_l\), has a higher weight. Intuitively, this kind of inference should not be allowed as it propagates wrong beliefs: The votes indicate that a negative decision should be made but instead, the positive information is used for the path computation and therefore propagated into the path scores. This may lead to false conclusions as shown in the next example.

**Example 3.1.12 (Dominating Votes).** In Figure 3.13(b), \(r_1\) is unequal to \(r_3\) which is positively connected to \(r_4\). Additionally, the crowd is unsure whether \(r_4\) is unequal to \(r_2\). Therefore, it should not be possible to infer whether \(r_1\) and \(r_2\) belong to the same or different entities. Computing the path scores as previously shown, MINMAX proposes a negative score \(n_{12}^* = 3\) for the record pair \((r_1, r_2)\) but as there exists no positive path between \(r_1\) and \(r_2\), \(p_{12}^* = 0\). As a result, the relationship of \((r_1, r_2)\) is falsely estimated.
Properties of MinMax

To counter the effect described in Example 3.1.12, we modify MINMAX to limit the propagation of dominated edges. More specifically, we can only compute a path between two records \( r_i \) and \( r_j \) if any edge on the path is not dominated by its opposite edge.

**Corollary 3.1.1 (Edge Domination).** A vote \( p_{ij} \) (\( n_{ij} \)) for a record pair \([r_i, r_j]\) can be used for path computation only if it dominates \( n_{ij} \) (\( p_{ij} \)), i.e., \( p_{ij} > n_{ij} \) (\( p_{ij} < n_{ij} \)).

Next to avoiding wrong propagation of values, this corollary also guarantees that the path computation of MINMAX adheres to a weaker form of (anti-)transitivity. Remember that when working with erroneous data, conflicting information may lead to interpretation inconsistencies, i.e., there exist paths between records \( r_i \) and \( r_j \) that convey contradicting decisions, one positive and one negative. Weak (anti-)transitivity guarantees that the decision function proceeds cautiously when encountering these noisy path values. Instead of risking a faulty decision, the decision function will return ‘unknown’ instead.

**Definition 3.1.15 (Weak Transitivity).** For records \( r_i, r_k, \) and \( r_j \) and any votes graph:

\[
f(r_i, r_k) = 'yes' \land f(r_k, r_j) = 'yes' \implies f(r_i, r_j) = 'yes' \lor f(r_i, r_j) = 'unknown'
\]

Weak anti-transitivity is defined analogously.

**Proposition 3.1.5 (Weak Transitivity of MINMAX).** MINMAX is weakly transitive.

**Weak Transitivity of MINMAX.** Given three records \( r_i, r_j, \) and \( r_k \in R \) transitivity is violated if \( f(r_i, r_k) = 'yes', f(r_k, r_j) = 'yes' \) but \( f(r_i, r_j) = 'no'. \) Path scores \( p^*_{ij} \) and \( n^*_{ij} \) are computed as follows:

\[
p^*_{ij} = \min\{p^*_{ik}, p^*_{kj}\} \quad (3.1)
\]

\[
n^*_{ij} = \max\left\{ \min\{n^*_{ik}, H^p_{kj}\}, \min\{H^p_{ik}, n^*_{kj}\} \right\} \quad (3.2)
\]

Equation 3.1 follows from the definition of positive path scores. Note that if there is any direct edge between \( r_i \) and \( r_j \), it can only improve the positive score \( p^*_{ij} \) due to the maximum computation of MINMAX, which makes Equation 3.1 the worst case scenario for positive path computation. Computing \( n^*_{ij} \) means evaluating each negative subpath going through \( r_k \) with all possible positive paths \( H^p \) that complete the path from \( r_i \) to \( r_j \). Any direct edge between \( r_i \) and \( r_j \) is already considered in this computation because it is...
reflected in the subpath scores. For example, the computation of \( n^*_{ik} \) already considers all paths between \( r_i \) and \( r_k \) including those that contain a direct edge between \( r_i \) and \( r_j \). If transitivity is violated then \( p^*_{ij} < n^*_{ij} \) must hold. However, this contradicts MinMax as described in the following case analysis designed according to Equation 3.1 and 3.2:

**Case 1:** \( p^*_{ij} = p^*_{ik} \)

- **Case 1.1:** \( n^*_{ij} = \min \{ n^*_{ik}, H^0_{kj} \} \). Under the transitivity violation assumption, this means that \( p^*_{ik} \leq n^*_{ik} \) which violates Corollary 3.1.1 and the initial assumption that \( f(r_i, r_k) = \text{‘yes’} \).

- **Case 1.2:** \( n^*_{ij} = \min \{ H^0_{ik}, n^*_{kj} \} \). This means that \( p^*_{ik} \leq H^0_{ik} \) for any positive path \( H^0_{kj} \). A strict inequality contradicts MinMax and the positive path score definition as \( p^*_{ik} \) is the strongest path between \( r_i \) and \( r_j \). An equality relationship is achieved for example if there is only one path or all paths have the same weight. As a result, it is possible to obtain \( p^*_{ij} = n^*_{ij} \), thus \( f(r_i, r_j) = \text{‘unknown’} \). This behavior is the reason, why MinMax only guarantees weak transitivity.

Therefore, weak transitivity cannot be violated if path scores are computed as in Definition 3.1.14 and if MinMax is used as a decision function.

**Case 2:** \( p^*_{ij} = p^*_{kj} \) The proof for this condition is analogous to Case 1.

**Weak Anti-transitivity.** For records \( r_i, r_k, \) and \( r_j \) and any votes graph:

\[
\begin{align*}
  f(r_i, r_k) &= \text{‘yes’} \land f(r_k, r_j) = \text{‘no’} \\
  &\implies f(r_i, r_j) = \text{‘no’} \lor f(r_i, r_j) = \text{‘unknown’}
\end{align*}
\]

Weak anti-transitivity can be proven in a similar fashion. Violating weak anti-transitivity (i.e. \( f(r_i, r_j) = \text{‘yes’} \) and therefore \( p^*_{ij} > n^*_{ij} \)) contradicts the MinMax definition as well as the assumption that \( f(r_k, r_j) = \text{‘no’} \). Even though MinMax adheres to a weaker variant of transitivity, we show experimentally that transitivity is in practice leveraged efficiently in Section 3.1.2.5.

Furthermore, compared to a simple majority-based decision function, i.e., a function that decides in favor of the majority for every record pair \( [r_i, r_j] \) (MA), MinMax obtains higher result quality due to this property. Quality in this context is defined as the pairwise accuracy of records within the entity resolution solution. For example, if \( r_i \) and \( r_j \) are in the same cluster according to the ground truth but are put into different clusters by
the ER algorithm then the accuracy of the result ER solution decreases. To understand why we claim that MinMax will eventually lead to the same or a better quality than a majority-based approach that builds upon dominating edges, let us revisit the running example in Figure 3.12. Here, we have observed one false decision between the baby animals $r_1$ and $r_3$. A majority-based algorithm would at this point form the decision ‘no’ and move on to the next record pair. However, the MinMax path scores in this example are $p_{(1,3)}^* = 3$ and $n_{(1,3)}^* = 2$ and if quorum $q_n$ is bigger than 1, then this record pair is marked as unresolved. As a result, the system would require more information from the crowd workers to form a decision. In the worst case, this decision would not change making the final decision equivalent to that of the majority approach. In the best case, new workers would distinguish the species and increase the number of positive votes.

**Corollary 3.1.2 (MinMax Result Quality).** The result quality of MinMax is at least as good as the result quality of a majority-based decision function.

More formally, under the assumption of non-malicious task workers, i.e., worker answers correctly with probability $p > 0.5$, the following observation holds: Given a vote set $E_i$ at timestamp $t_i$ and an enriched vote set $E_{i+1}$ at $t_{i+1}$ with $|E_i| < |E_{i+1}|$, the decisions made by MinMax at $t_{i+1}$ are more accurate than at $t_i$. Imagine that at $t_i$, a majority-based approach MA and MinMax compute their ER solution. Either decision made by MinMax is then the same as made by MA or ‘unknown’ due to the weak transitivity of MinMax. To resolve unknown decisions, MinMax enriches the edge set to $E_{i+1}$. The ER solution of MinMax based on $E_{i+1}$ has at least the same quality as for $E_i$ because $p > 0.5$. As a result, the decisions made by MinMax at $t_{i+1}$ are at least as good as the decisions made by MA at $t_i$, i.e., MinMax might incur higher cost but will not have worse quality than MA.

**Discussion of MinMax**

MinMax is a decision function computed on absolute values, associating record pairs with at most two numerical values that describe the relationship of the records in this pair. In contrast to working with absolute values, relative decision functions, i.e., computing the score of a record pair based on their distance and connectivity, can be used. The drawback of these functions is that they require to compute and maintain all paths between record pairs which is inefficient to execute in practice, especially in highly connected graphs. Furthermore, they have to be selected carefully as they need to fulfill the graph
properties described in Section 3.1.2.2. This cannot be guaranteed for functions such as \textit{sum} and \textit{count} for example, [GKRW12]. Probabilistic decision functions are another set of functions that have been explored in [WLGM13] amongst others where the authors also show that they are infeasible to compute even with a perfect crowd. For the remainder of this part of our work, we will thus focus on MINMAX as a representative of a group of decision functions that adhere to the presented graph properties and provide minimal computational overhead.

### 3.1.2.3 ER Algorithm

Traditional ER algorithms use the pair-wise information in a graph to construct a clustering that represents the final ER solution. This methodology is defined in the framework algorithm (Algorithm 1, Section 2.2.3) as the data interpretation problem. In the last section, we have discussed how we can find pair-wise information. Now, we examine how we can efficiently cluster records into entities. Example algorithms herefore are cut or correlation clustering, [BBC04], which we adapt in the following to suit the incomplete resolution space that is inherent to incremental ER. Before going into detail on the applied ER algorithm, we first establish how the votes graph and the MINMAX matrix can be adapted when new votes are collected from the crowd. The observations that we make are specific to MINMAX but can be easily adapted to any decision function that has the properties defined in Section 3.1.2.2.

#### Updating the Votes Graph

As explained in Section 2.2.3, incremental crowdsourced ER requires an adjustment of its queue and its ER solution whenever new information becomes available. To understand
the ER update process, i.e., the integration of a new positive or negative vote, first imagine
how an update is propagated: Obviously, an update affects the records that are directly
modified. From there on, it may affect connected records, which then further propagate
the changes through the votes graph. This notion is visualized in Figure 3.14 for the
running example where an edge between records $r_3$ and $r_4$ is modified while the edges
between $r_1$ and $r_3$ resp. $r_2$ and $r_4$ remain untouched. As a result, the following three
update operations need to be explored:

1. All paths connecting $r_3$ to any other record through $r_4$ may be modified.

2. All paths connecting $r_4$ to any other record through $r_3$ may be modified.

3. All paths that are now connected because of the new edge between $r_3$ and $r_4$ may
be modified.

The records that are (indirectly) modified by the changes are called the update component
of the update. They are those records for which either the positive or the negative path
connecting them to any other record has been modified directly or indirectly through the
update of an edge.

**Theorem 3.1.7** (Update Component). The changes due to a new or modified edge $e_i \in E$
need to be propagated only along those edges $e_j \in E$ for which either $p^*_ij$ or $n^*_ij$
changes.

**Proof.** The validity of this theorem derives from the fact that an update is a local change
that affects the global computation of paths by modifying their subpaths. Therefore,
if a path is not changed, then the scores of other paths that use this subpath will not
change. The propagation of updates along edges is analogous to the propagation of change
described in Section 3.1.1.

The update functionality used to propagate changes for MinMax is described in Algo-
rithm 3. Here, the increment computation is split into two parts: Algorithm UPDATE
describes how an increment can be realized, analogous to what is visualized in Figure 3.14;
Function COMPUTE then propagates the update along all affected paths recursively. The
actual update computation uses a boolean path marker $\gamma$ for declaring negative and pos-
itive paths (Lines 8 - 22) as follows:

- **If** $\gamma = false$, the algorithm knows that the current path is negative. It therefore
evaluates only positive outgoing edges for nodes that have not been traversed on
this path (Line 10). For each of these candidate paths, COMPUTE is called again to propagate the update further.

- If \( \gamma = \text{true} \), the algorithm follows the same procedure as above but has to consider all outgoing edges, independent of whether they are positive or negative. As a result, it is recursively called for both scenarios (Lines 20-22).

The COMPUTE function is used by Algorithm UPDATE for all three update scenarios described previously. The first two scenarios (Lines 3 and 4) explore all modified paths in the respective update direction. Solving the third scenario (Line 5) is more complex as all path alternatives that provide the MINMAX score for a record pair \((r_i, r_j)\) need to be stored. As this is infeasible to compute, only the path between \(r_i\) and \(r_j\) that has the highest path score, referred to as \(\text{path}_{(r_i,r_j)}\), is stored. In case there exist multiple paths with equal scores, the shortest path is chosen. Obviously, this approximation allows for impreciseness in the result but as we show in our experimental evaluation, this simplification has no discernible impact on the result quality. After the changes to the MINMAX matrix have been computed, we adjust the current ER solution by running a clustering algorithm on the set of modified records and their corresponding clusters (Line 7).

**Example 3.1.13 (UPDATE Algorithm).** *Going back to the initial example (Figure 3.13(a)), imagine that the edge between \(r_3\) and \(r_4\) is newly inserted. The algorithm then first explores the positive path between \(r_3\) and \(r_2\) but as \(p^*_{(2,3)}\) is already 1, it is not further pursued. In contrast, a negative path connecting the records did not previously exist, thus \(n^*_{(2,3)}\) is set to 3. Following the connecting positive edge to \(r_1\), the algorithm then sets \(n^*_{(1,3)}\) to 3. As no path value connecting \(r_4\) to any other record via \(r_3\) is changed, the update process finishes.*

**Theorem 3.1.8 (Complexity of UPDATE).** *Updating the MINMAX matrix has a complexity of \(O(m^2)\) in the worst case. Here, the variable \(m\) represents the number of records in the update component.*

**Proof.** Updates connecting records \(r_i\) and \(r_j\) are propagated in three different ways as described previously. The number of paths in the update component is maximized if the update affects \(m/2\) in the first and second propagation phase (Lines 3 and 4). In the third phase, at most \(m^2/2\) pair-wise scores need to be updated which results in a quadratic worst-case execution time.

Given the cost-optimized incremental structure of the votes graph, the average execution time is significantly lower in practice.
3.1: Entity Resolution

Clustering Records

Using up-to-date positive and negative path scores, an ER solution based on the current state of the votes graph and MIN MAX matrix can be computed. For that purpose, we employ a variant of correlation clustering which adheres to the concepts of intra-cluster density and inter-cluster sparsity. To evaluate whether two records \( r_i \) and \( r_j \) refer to the same entity, the benefit and penalty of having them in the same cluster are weighed according to their \( p_{ij}^* \) and \( n_{ij}^* \) scores: If \( p_{ij}^* > n_{ij}^* \) then it is more beneficial to assign both to the same entity, otherwise they are assigned to different entities. We use a variation of an established approximation for clustering records called cautious correlation clustering [BBC04] (Algorithm 5). The Resolve algorithm proceeds as follows: It first randomly selects a record \( r_1 \) that has not been assigned to a cluster. It is then added to a new cluster \( c_1 \) together with all those records \( r_j \) that it is positively connected to, i.e., \( p_{ii}^* > n_{ij}^* \). Record \( r_j \) can only become a part of \( c_1 \) if it is not a part of a cluster yet. It next initiates a vertex removal phase in which all those records are removed from \( c_1 \) where the penalties of keeping them in \( c_1 \) outweigh the benefits (Line 7). For example if a \( r_j \) has a positive relationship with weight 1 to \( r_i \) but a negative relationship with weight 2 with \( r_k \in c_1 \), it is better to remove \( r_j \) from \( r_k \) if the score of fulfills \([r_i, r_k] > 1\). The ‘goodness’ of record \( r_j \) is examined in function isGood which is a weighted validity computation analogous to the computation of \( \delta \)-goodness in [BBC04]. After the vertex removal phase has been completed, a record addition phase is initiated. Here, any records are added to \( c_1 \) if the benefit of adding that record outweighs the penalty (Line 11).

Example 3.1.14 (Resolve Algorithm). In Figure 3.13(a), the votes graph contains records \( R = \{r_1, r_2, r_3, r_4\} \). Picking \( r_1 \) randomly, \( r_1 \) is added in addition to \( r_2 \), with \( p^*_{1,2} > n^*_{1,2} = 3 > 1 \), to cluster \( c_1 \). Removing either \( r_1 \) or \( r_2 \) from the cluster does not provide a better clustering score, thus the cluster remains unchanged. Neither \( r_3 \) nor \( r_4 \) are added to \( c_1 \) as for both the negative path values to any record in \( c_1 \) is at least as high as the respective positive scores. Similarly, the algorithm constructs cluster \( c_2 \) containing both \( r_3 \) and \( r_4 \). Thus, the final clustering is \([\{r_1, r_2\}, \{r_3, r_4\}]\).

Instead of running Resolve on all records after an update, it is only run on those records that are in any cluster that has been touched by an update (Algorithm 3, Line 7). Since these records do not necessarily represent whole clusters, the algorithm first expands the update component into a transitive update component, i.e., the original update component and all records that are in any of the clusters contained in the update component. Partial
Chapter 3: Data Integration Methodology

ER clustering for the transitive update component is triggered after each update to the votes graph to provide a consistent ER solution.

**Theorem 3.1.9 (Complexity of \textsc{resolve}).** Running entity resolution on the updated parts of the \textsc{MinMax} matrix has a worst case complexity in $O(|R|m^*)$.

**Proof.** The vertex removal phase requires $m^*$ repetitions at most where $m^*$ is the number of records in the transitive update component. The vertex addition phase is bounded by the number of overall records in the graph, $|R|$. As both phases are initiated for all $m^*$ records and $m^* < |R|$, \textsc{resolve} runs in $O(|R|m^*)$. \hfill \qed

**Complexity**

In practice, the \textsc{update} as well as \textsc{resolve} algorithm perform near linear when scaling up the number of nodes in the votes graph. To demonstrate the scaling behavior, we examined the processing time, i.e., the end-to-end time spent on integrating an update, for a synthetic dataset as shown in Figure 3.15. The dataset has a maximum entity size of 50 and the distribution of entities within the ground truth follows a Zipf distribution. Varying the number of records, we can see that the processing time increases as expected due to the bigger votes graph which entails more propagation of updates. The increase is not quadratic but approximately linear: For $n = 100$ the time spent on one update is 4.99ms, for $n = 250$ 11.95ms, for $n = 500$ 39.06ms, and for $n = 1000$ ms. The reason is that the graph size is not the only influence on execution time. Entity size as well as the amount of noise in the votes graph play a decisive role for our algorithm’s performance.
The higher the number of positive votes in the graph, the higher the number of positive or negative paths in the votes graph. As a result, processing time per update will increase. In different synthetic experiments we also observe that the computational effort per update is not the bottleneck for crowdsourced ER. Instead, experiments where clusters are very small usually take a lot of time to execute end to end. The reason is that even though the time spent on one update is small, a lot more updates are required to complete the votes graph and to find a path between all records. For a discussion about the processing time of our real-world datasets please refer to Section 3.1.2.5.

Path Optimization

While constructing the MinMAX matrix, the UPDATE algorithm described above maintains the current path and its score. To enable task requests efficiently, this knowledge can be used to define the investment points for further requests: If the score of a path needs to be modified (i.e., to increase certainty in the result), the minimal edge(s) in that path are those record pairs that have to be answered by the crowd. We observe that a minimal edge in a path between records $r_i$ and $r_j$ can most likely be improved if it satisfies either of the following requirements:

- If the path that is currently inspected is positive, then $p_{ij} \geq n_{ij}$ must hold.
- If the path is negative, then $n_{ij} \geq p_{ij}$ must hold.

The intuition here is that an edge can only get stronger if it gets more votes in its favor. This is more likely if it is already a dominating edge because if it is dominated, the crowd so far thought this decision to be the wrong one for this record relationship. Asking more questions for that record pair will thus likely result in the strengthening of the opposite decision which will not enhance the certainty of the candidate pair in question.

Example 3.1.15 (Edge Selection). Observing the paths connecting records $r_2$ and $r_3$ in Figure 3.13(a), there are two possible minimal paths, $\{r_3, r_1, r_2\}$ and $\{r_3, r_4, r_2\}$. The first path has its minimal edge between records $r_1$ and $r_3$ and since this edge is not dominated, improving the path at minimal cost would result in a request for more information on that record pair. The improvement of the second path on the other is not feasible as the minimal edge between $r_2$ and $r_4$ is dominated by its counterpart.
Chapter 3: Data Integration Methodology

3.1.2.4 Next-Crowdsource Problem

As explained in Section 3.1.2.1, the next-crowdsource problem is to decide which record pairs should be resolved next to improve result quality for the lowest possible additional budget. Whichever task, i.e., comparison between records, is asked next is dependent on the current state of the votes graph. If there is no knowledge on how any records are related, there cannot be an informed decision on which task to issue to the crowd. On the other hand, if there is some pairwise information available it can be leveraged to refine the current ER solution.

In our work, we assume independence of data preprocessing but can leverage a preprocessing step analogous to the steps described in [WLK+13, WLGM13]. Please refer to the publications dataset in Section 3.1.2.5 as an example. Note however that preprocessing steps such as hint generation through automatic similarity computation or blocking mechanisms that reduce the search space are an orthogonal problem to this line of work. The reason is that automatic similarity metrics commonly rely on some kind of syntactic similarity that is not necessarily equivalent to what humans perceive. For example if pictures show a well-known landmark from different angles, humans can use their knowledge of the building to answer the task based on semantic knowledge. Thus, we only focus on enhancing the already existing votes graph through human knowledge in this section.

Solution Space. Two alternative approaches to solving the next-crowdsource problem through a queuing system are discussed in Section 3.1.2.1: The first one generates all candidate pairs, monotonically requesting pairs, and terminating when all pairs have been asked or inferred. The second approach generates candidate pairs, requesting them iteratively while regularly checking whether other record pairs should be reinserted. Thus, it proceeds non-monotonically. For each of these approaches, a specified decision function provides information about record relationships. If there exist uncertain record pairs, the next-crowdsource component needs to decide a) in which order uncertain pairs should be addressed and b) how much budget should be allocated per candidate pair. Furthermore, when working with the crowd execution time is always a factor as outsourcing tasks to humans lowers the overall end-to-end time especially if the applied algorithm proceeds sequentially. This makes parallelization techniques and how they impact result quality and execution time an important alley to explore. To address both of these challenges, the next section details three querying strategies that can be used for record pair ordering. We introduce two types of parallelism for incremental ER and discuss their benefits and
disadvantages in comparison to sequential task execution.

### Querying Strategies

We call *querying strategies* those budget allocation strategies that determine the ordering of unresolved candidate pairs in the priority queue. To determine the ordering of candidate pairs, they use the consensus measure $\varphi_{ij}$ of two records $r_i$ and $r_j$. The consensus of a record pair $[r_i, r_j]$ describes how a decision function $f$ estimates the relationship of $r_i$ and $r_j$ numerically. More specifically, it is a potentially enriched output of $f$. In Section 3.1.2.2, we discussed decision functions that map the relationship of $r_i$ and $r_j$ to ‘Yes’, ‘No’, and ‘Unknown’ decisions. If $\varphi_{ij} \in [-1, 1]$, these correspond to 1 (‘Yes’), -1 (‘No’), and 0 (‘Unknown’). Given more expressive decision functions such as $f_M$ or majority-based decision mechanisms, the consensus measure can be adapted: For example given a quorum of 3, $p^*_i = 3$, and $n^*_ij = 1$, $\varphi_{ij}$ is 0.67. Other decision functions that can be employed here are normalized distance-based metrics or adjacency-based mechanisms that decide on the consensus based on the relationship of a record to its neighbors such as [VBD14]. Querying strategies then use the consensus of record pairs to prioritize record pairs over others.

**Definition 3.1.16 (Querying Strategy).** The querying strategy $\omega$ of an ER process determines the order in which candidate pairs are evaluated. For that purpose, it uses the consensus measure $\varphi_{ij}$ to determine the relative position of a record pair $[r_i, r_j]$ in $Q$.

Any querying strategy applied in the context of ER realizes some trade-off between the budget spent and the quality that it wants to achieve. However, strategies may vary on the exact trade-off and whether they favor quality over cost or vice versa. In general, querying strategies are similar to active learning [Set12] because we try to learn the results by actively asking more questions.

The first strategy that we present next aims to maintain consistency within the part of the dataset that the algorithm has already looked at. It is therefore designed to verify that the decisions made so far are in fact correct. We refer to this querying strategy as the **error reduction** strategy (ErS).

**Definition 3.1.17 (ErS).** Given record pairs $[r_i, r_j]$ and $[r_k, r_l]$, query strategy $\omega_{ErS}$ prioritizes $[r_i, r_j]$ over $[r_k, r_l]$ if

$$|\varphi(r_i, r_j)| > |\varphi(r_k, r_l)|$$

and if for both record pairs $|\varphi(r_i, r_j)| \neq 1$ respectively $|\varphi(r_k, r_l)| \neq 1$ holds.
This strategy is able to provide high precision ER solutions: Instead of investing the budget into solving a lot of different pairs only partially, it focuses on resolving edges before moving on to the next record pair. As a result, this strategy provides introduces votes into the graph that it is certain about. In contrast, the second strategy that we introduce promotes completeness over correctness and distributes the budget in a breadth-first manner. It allows for a fast (but possibly imprecise) initial representation of record relationships within the dataset. We will refer to this strategy as the uncertainty reduction strategy (UrS) as it tries to obtain some information for any record pair first before asking more questions in depth to become more certain the record relationships.

Definition 3.1.18 (UrS). Given record pairs \([r_i, r_j]\) and \([r_k, r_l]\), query strategy \(\omega_{urS}\) prioritizes \([r_i, r_j]\) over \([r_k, r_l]\) if

\[|\varphi(r_i, r_j)| < |\varphi(r_k, r_l)|\]

and if for both record pairs \(|\varphi(r_i, r_j)| \neq 1\) respectively \(|\varphi(r_k, r_l)| \neq 1\) holds.

As we assume uncertainty to be at its highest when \(\varphi(r_i, r_j) = 0\), the absolute distance of the consensus measure to 0 is the certainty of \([r_i, r_j]\) for \(f\). With no prior knowledge of record pairs, \([r_i, r_j]\) = 0 holds. Record pair \([r_i, r_j]\) would therefore be prioritized in comparison to any other record pair \([r_k, r_l]\) for which \(0 < |\varphi(r_k, r_l)| < 1\) holds. The reason is that such a score would indicate some knowledge of \([r_k, r_l]\).

Hybrid strategy (HS). Under the assumption that the crowd fulfills their tasks perfectly, UrS will always dominate ErS in terms of its quality gain per cost unit as shown in Figure 3.16(a) on a dataset consisting of landmarks in Paris, France, and Barcelona, Spain with a perfect synthetic oracle. Quality is captured here through the f-measure of the ER solution while one cost unit corresponds to one evaluated record pair, i.e., one crowdsourced task. For further details about the dataset and the experimental setup please refer to Section 3.1.2.5.

In real-world use cases, the assumption of a perfect oracle does not hold which causes the dynamics of the two querying strategies to shift: The quality improvement of ErS is predictable albeit at a higher overall cost while UrS makes an increased number of false local decisions especially if only little budget has been invested. These issues can be addressed through a hybrid querying strategy that combines the best of both. As positive task decisions have more impact on the ER solution due to the way (anti-)transitivity is defined, the hybrid strategy resolves positive decision candidates cautiously, following
3.1: Entity Resolution

![Graphs showing F-Measure vs Crowd Accesses](image)

(a) perfect crowd  
(b) \( f_n = f_p = 0.1 \), sequential  
(c) \( f_n = f_p = 0.1 \), parallel

Figure 3.16: Queuing strategies for Landmarks dataset (synthetic crowd).

ERS. Candidate pairs that potentially lead to negative decisions are processed according to URS to avoid spending budget on a request that potentially has not a lot of impact. This differentiation of decision strategies is analogous to the idea of reward functions in active learning: A positive vote indicates that the task is an important one and should be prioritized. In contrast, a negative vote would not trigger a reward and the record pair would be crowdsourced again at a later point again.

**Definition 3.1.19 (Hybrid Strategy).** Given record pairs \([r_i, r_j]\) and \([r_k, r_l]\), query strategy \(\omega_{HS}\) prioritizes \([r_i, r_j]\) over \([r_k, r_l]\) if

\[
\varphi(r_i, r_j) > \varphi(r_k, r_l)
\]

and if for both record pairs \(|\varphi(r_i, r_j)| \neq 1\) respectively \(|\varphi(r_k, r_l)| \neq 1\) holds.

If \(\varphi(r_i, r_j)\) and \(\varphi(r_k, r_l)\) are positive, the strategy will thus choose the record pair that has less uncertainty but is not yet certain following \(\omega_{ERS}\). In contrast, if \(\varphi(r_i, r_j)\) is below
0, it will only be chosen over $\varphi(r_k, r_l)$ if its distance to 0 is lower which is the case if its consensus measure is bigger.

To enable a seamless hybrid strategy, it can be implemented through a double queue system: One of the queues contains positive candidate pairs and applies $\text{E}r\text{s}$ while $\text{U}r\text{s}$ is used for the negative queue. Whenever a record pair $[r_i, r_j]$ is inserted into this hybrid queueing system, it is inserted into either the positive or negative queue based on the current state of $p_{ij}^*$ and $n_{ij}^*$. For record pair extraction, the positive queue is accessed first and a record pair is pulled from the negative queue only if the positive queue is empty.

**Workload Parallelization**

Integrating the crowd into the information collection process comes not only at a monetary but also temporal cost. In fact, it increases response time drastically. This is why task parallelization has become an established technique to limit the increase in execution time. Generally, there are two types of parallelization that are applicable for this kind of pair-wise crowdsourcing, *intra-task* and *inter-task* parallelization.

**Intra-task parallelization.** Instead of asking for a record pair once, the same pair is issued multiple times to different workers. For example for quorum or majority-based techniques, it is possible to compute the required number of tasks to achieve certainty on the fly: It depends on how many answers are required and how many tasks have been returned for this record pair previously.

**Inter-task parallelization.** Instead of asking for a series of distinct record pairs sequentially, the required pairs are analyzed and if they found independent of each other, they
are released onto the crowdsourcing platform in parallel within a batch. This technique has been discussed in prior work, [WLK+13], and is commonly realized by generating a spanning tree over all entities. In practice, we point out that uncertainty of the crowd hinders inter-task parallelization as shown with the following example.

**Example 3.1.16** (Inter-task Parallelization). *In Figure 3.17, \([r_3, r_4]\) is already known after which the algorithm decides to parallelize tasks \([r_1, r_2]\), \([r_1, r_3]\), and \([r_2, r_4]\). The answers to these tasks causes uncertainty to arise between \(r_1\) and \(r_3\) resp. \(r_2\) and \(r_4\).*

When comparing sequential and parallel task execution for the different querying strategies with noisy answers occurring with a likelihood of 10%, (Figure 3.16(b) respectively Figure 3.16(c)), the following three observations can be made: First, the absolute cost for ErS decreases when parallelizing task execution. The reason herefore is that inter-task parallelization requires connecting previously unconnected entities in a manner aimed to provide maximal coverage. To generate the spanning tree, the algorithm iterates over all candidate pairs in the queue in descending order of their objective. As a result, it is more likely that positive rather than negative candidate pairs are part of the spanning tree which in return improves the performance of ErS. The second observation is that the cost for both UrS and HS increases. Here, the opposite effect as for ErS takes hold: Parallelizing them forces both strategies to execute tasks with lower priority within the current batch if other (higher priority) tasks are not consistent with the existing spanning tree. Additionally, the resolution of contradictions in the result set incurs higher cost because of the necessity for more information but at the same time it lowers result quality due to noisier intermediate results. Third, HS maintains a better result cost and quality trade-off than either alternative approach because it is still able to leverage the benefits of both ErS and UrS.

### 3.1.2.5 Experimental Evaluation

In this section, we compare fault-tolerant data interpretation mechanisms with consensus-based approaches and discuss the different querying strategies and parallelization techniques introduced in the previous section.
Experimental Setup

Two different real-world datasets that contain pictures resp. publication records are used in this evaluation. We evaluated these datasets both with synthetic and real-world crowds. To provide an extensive experimental evaluation on crowdsourced data, the datasets were crowdsourced completely on the Amazon Mechanical Turk platform, i.e., ten crowd workers solved each possible record pair. Each of these workers has a 90% acceptance rate to avoid malicious workers. For all the experiments shown here, the results of the respective setup are averaged over at least 100 different single experiments. Note that this methodology allows us to compare the algorithms in a robust setting where all of them have the same chance to succeed.

Landmarks dataset. This dataset consists of 266 pictures of landmarks in two European cities, namely Paris, France, and Barcelona, Spain. It is based on a picture classification dataset for visual object identification algorithms, [AKTS10], and contains 13 entities that are landmarks such as the Arc de Triomphe. In each task, a crowd worker is shown a pair of pictures and decides whether they show the same landmark. As shown in Table 3.7, there exist a total of 352,450 record pairs out of which about 7.8% are correctly identified as belonging to the same landmark. The crowd also identified another 4.6% as positive matches even though the ground truth was negative. As a result, 37.1% of all positive answers should be in fact negative. In total, about 2.8% of all extracted answers are false negative answers and 84.8% of all answers were true negatives.

Publications dataset. This dataset is a compressed version of the CORA dataset which is commonly used for string similarity evaluation 3. It is derived from the original dataset through relative compression, i.e., if an entity contained 5% of the records in the original dataset, it will contain approximately the same percentage of randomly selected records in the smaller dataset. Overall, this dataset has 3.47 records per entity on average and thus results in a higher amount of candidate negative decisions than the landmarks dataset. In fact, only 2.42% of all decision pairs are positive, out of which 64.04% are identified as such. We observe a high ratio of false positive decisions to the overall number of positive decisions: Here, 36.2% of all positive decisions are erroneous. We also observe that the absolute number of false positive and false negative decisions in the answer set are about equal and overall lower than for the landmarks dataset.

3http://secondstring.sourceforge.net/
## 3.1: Entity Resolution

<table>
<thead>
<tr>
<th>Data</th>
<th>Landmarks</th>
<th>Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>#records</td>
<td>266</td>
<td>198</td>
</tr>
<tr>
<td>#rec. pairs</td>
<td>35,245</td>
<td>19,503</td>
</tr>
<tr>
<td>#entities</td>
<td>13</td>
<td>57</td>
</tr>
<tr>
<td>avg #rec./ent.</td>
<td>20.46</td>
<td>3.47</td>
</tr>
<tr>
<td>max #rec./ent.</td>
<td>43</td>
<td>14</td>
</tr>
<tr>
<td>min #rec./ent.</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistics</th>
<th>absolute</th>
<th>absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>#true positives</td>
<td>27,512</td>
<td>3,023</td>
</tr>
<tr>
<td>#true negatives</td>
<td>298,822</td>
<td>188,592</td>
</tr>
<tr>
<td>#false positives</td>
<td>16,248</td>
<td>1,718</td>
</tr>
<tr>
<td>#false negatives</td>
<td>9,868</td>
<td>1,697</td>
</tr>
</tbody>
</table>

Table 3.7: Datasets Overview

**Algorithms.** We evaluate three different data interpretation algorithms from the core classes identified in Section 3.1.2.1.

- **CER.** As state-of-the-art consensus-based strategy we choose CrowdER, [WLK+13], which uses a majority-based decision strategy. It monotonically requests all candidate pairs from the crowd, merging those that are connected by a positive decision and can thus be classified as a consensus-based decision strategy.

- **FER.** This algorithm represents the class of fault-tolerant strategies. It incorporates the same non-repetitive queuing mechanism as CER but uses MINMAX as decision function with a quorum $q = 3$ and repeats each task until $q$ or the edge budget $(b_E=10)$ is reached. Experiments with other quorum values are omitted here but show similar trends.

- **FEER.** This is an exhaustive fault-tolerant strategy which implements FER but maintains a non-monotonic queuing system: If an update to the internal MINMAX strategy causes a pair to become uncertain, it is inserted back into the queue.

This evaluation will also focus on the two task ordering techniques discussed in Section 3.1.2.4: queuing strategies and parallelization. We evaluate the three queuing strategies presented previously, ERS which is a cautious mechanism that aims to optimize intermediate results, URS which internally orders its candidate pairs according to their level
of uncertainty, and their hybrid HS which cautiously processes positive and optimistically processes negative candidates. The hybrid strategy will serve as default strategy if not otherwise declared. To assess the impact of parallelization techniques, we implement as baseline a sequential process that iteratively asks for exactly one record pair. We compare it to a combination of inter- and intra-task parallelization, i.e., if parallelized, the algorithm automatically computes a spanning tree over all entities and within an entity as well. For all candidate pairs, it then computes the minimal necessary investment to reach quorum or consensus and issues the resulting record pairs as a batch.

**Metrics & Implementation.** This evaluation uses mainly two standardized metrics, quality and cost. The cost of any experiment is simply measured as the number crowd accesses which is equivalent to the amount of requested record pairs. To measure quality, we use precision and recall where precision is the percentage of record pairs that are correctly associated with the same entity and recall is the percentage of record pairs that we correctly assign to the same entity. To provide a unified quality metric, the $f$-measure of these values is used as standard quality metric, defined as $\frac{P \cdot R}{P + R}$. We implemented all of the presented algorithms and strategies in Java, and experimented on a Linux machine with eight Intel Xeon L5520 cores (2.26GHz, cache 24MB).

**The Impact of Crowd Error**

To exemplify the impact of errors made by crowd workers, we synthetically generated two types of noise for the landmarks dataset. The first noise is false positive noise, $f_p$, which describes the percentage of decisions where crowd workers wrongly classify records $r_i$ and $r_j$ to belong to the same entity when in fact they belong to different entities according to the ground truth. Analogously, the second type of noise is false negative noise, $f_n$, where $r_i$ and $r_j$ are falsely assigned to the same entity. Figure 3.18 and Figure 3.18a show the results of this set of experiments, comparing different data interpretation models under uniform synthetic noise varying both $f_n$ and $f_p$. We observe that if the crowd answers perfectly (Figure 3.18a), $Fer$ and $Feer$ show rapid quality improvement per cost unit. $Feer$ outperforms $Fer$ because it dynamically adjust the priority queue, pushing potentially positive candidate pairs to the front of the queue and therefore maximizing the information gain per crowd access. The cost/quality trade-off for $Cer$ is predictable and is correlated to the number of votes $v$ requested: As the crowd answers perfectly, a smaller $v$ means having a comparably better quality result for a lower budget. When introducing
noise into the experimental setup, we make the following three observations.

**False Positive Information.** This is the type of crowd error that has the most impact on output quality as seen in Figure 3.18b: In CER, transitive closure is applied for entity resolution which means that a positive decision leads to a merge of records into one entity. As these decisions are never questioned, the recall and precision of CER (Figure 3.19) soon decrease as records are merged that belong to different entities. Generally, we observe that recall increases over time while precision (i.e., records correctly belong to the same entity) decreases over time when generating false positive noise. FEER and FER recover from an initial drop of precision because they question their resolutions whenever they encounter a contradiction. This can be observed in Figure 3.19a through the increase in the respective precision curve for FEER (FER) after approximately 3k (4k) crowd accesses.

**False Negative Information.** If the crowd answers with false negative answers (Fig-
ure 3.18c), the workers decide to keep records apart which should be in the same entity with a likelihood of $f_n = 0.3$. We observe in this set of experiments that an increase in the number of questions asked ($v$) leads to a provable increase in quality for CER. Here, with $v = 9$ instead of 5, the final ER solution reaches a f-measure of 0.69 instead of 0.12. Compared to FEER and FER, CER still provides a lower quality improvement over crowd accesses because in order to reach better quality, more budget would need to be invested into finding an ER solution. We notice also in this experiment that varying $f_n$ changes the recall of the results but never the precision in contrast to false positive information which influences both. To understand this behavior, remember the definition of precision and recall: Precision is negatively influenced through records being falsely assigned to the same entity, recall is negatively influenced by records in different entities that should be in the same entity according to the ground truth. As records can never falsely belong to the same entity through false negative votes, precision is not influenced by this type of crowd error.

Noisy Crowd Information. Combining both types of crowd error leads to a visible quality decrease for both CER variations and larger budget requirements for both FEER and FER (Figure 3.18). We again observe that increasing the budget for CER in fact positively influences the result quality but as we will show in our real-world experiments, it never reaches the same level of quality as both fault-tolerant approaches. The additionally required budget investment for these can be explained through the decision behavior of MinMax: If uncertain, it requests more information from the crowd, thus allocating more budget. If the budget is granted, it leads to a steady improvement in result quality which
itself is better than the CER result quality at any point in time.

**Landmarks Results**

The landmarks dataset is an interesting use case as it provides an environment where no similarity metric enhances the performance of the ER strategies. Here, every candidate pair is initially equally likely. To contrast our techniques, we now compare the fault-tolerant mechanisms with the traditional consensus-based approach using the same crowdsourced results obtained by posting tasks on Amazon Mechanical Turk for all approaches. Additionally, we evaluate the different queuing strategies after which we highlight the advantages and disadvantages of parallelization.

**Data Interpretation**

To compare the quality of fault-tolerant mechanisms to algorithms assuming a perfect world, the first set of experiments compares the cost/quality trade-off for CER, Fer, and Feer (Figure 3.20).

**Performance of CER.** Figure 3.20a shows the development of all data interpretation mechanisms in terms of quality over crowd access. The low performance of CER in comparison to Feer and Fer (see difference in quality $\Delta q$ and cost $\Delta c$ in Figure 3.20a) is due to its sensitivity to false negative decisions which form 26.4% of all candidate positive decisions in the landmarks dataset. While CER maintains a minimum of 0.96 precision over time, its recall is at most 0.21 even if the number of votes considered $|v|$ is increased from 5 to 9 which only results in a decrease in cost/quality gain. At this point, recall that a) positive decisions reduce the search space of the algorithm and b) the landmarks dataset has only 13 big entities. Uncontested false negative decisions therefore have significant impact on result quality.

**Performance of Fer and Feer.** As shown in Figure 3.20a, it is possible that Fer outperforms Feer. While Feer uses an adapting queuing system that obviously reduces the overall cost, Fer requests candidate pairs as long as its queue is not empty. Its monotonic queue is generated in the beginning and every candidate pair is polled from the queue in random order exactly once. Given noisy answers from the crowd, this mechanism actually improves result quality intuitively because it is not aimed to minimize the cost but to ask every candidate pair at some point in time which is more costly but also more
exhaustive than optimizing the queuing system. To verify this hypothesis, we implemented a modification of FEER that allows us to vary the connectivity of the entities associated with each record $\kappa$ which varies the number of record pairs per entity combination. A higher value of $\kappa$ resembles more record pairs that are requested to test the relationship of two entities. For example, if the first entity contains four records and the second entity contains three records, there exist a total of 12 record pairs. Instead of selecting one at random, it will select two random pairs if $\kappa$ is set to 0.2. Increasing the connectivity of the entities has immediate consequences: First, the cost of FEER increases and second, the quality of FEER improves significantly for this dataset as shown in Figure 3.20b. With $\kappa$ set to 0.2, we now observe a better quality to cost ratio for FEER (result quality of 0.916 with a total cost of 23,918 cost units) than even FER can offer (result quality of 0.856 with a total cost of 27,197 cost units).

Finally, Figure 3.21 shows the performance comparison of our algorithms for this dataset.
in sequential mode without any modifications to the connectivity. We observe that in this setup \( \text{FER} \) takes at most 8.12ms per update while the update cost of \( \text{FEER} \) is 51.93ms. The reason why \( \text{FEER} \) is slower than \( \text{FER} \) lies in the readjustment of the queuing system: As the dataset contains some noise, record pairs get occasionally reinserted into the queue. Generating these new record pairs and adjusting their position in the queue incurs computational overhead which is reflected in the execution time.

**Summary.** Figure 3.20a and Figure 3.20b show that for this dataset \( \text{FEER} \) reaches a better cost/quality trade-off faster than any other approach if the connectivity parameter \( \kappa \) is adjusted. Furthermore, all fault-tolerant strategies significantly outperform \( \text{CER} \) in terms of quality as we observe a minimal difference of at least 0.4 on the f-measure. This improvement can be reached at lower cost than needed for \( \text{CER} \) for \( \text{FEER} \). We also show that the low quality performance of \( \text{CER} \) does not depend on its available budget as the output quality only minimally increases with a higher budget (Figure 3.20a).

**Querying strategies**

When comparing the three proposed querying strategies for \( \text{FEER} \) in Figure 3.20c, similar cost/quality trade-offs in the real-world dataset than estimated with the synthetic crowd (Figure 3.16) can be observed: Here, \( \text{HS} \) clearly outperforms both alternative strategies as it provides better result quality at lower cost. The exact improvement that \( \text{HS} \) may offer depends on the underlying structure of the dataset and the crowd: More noise decreases the performance of \( \text{URS} \) and highlights the robustness of \( \text{HS} \). In comparison to \( \text{ERS} \), \( \text{HS} \) provides better cost/quality trade-off if the number of entities in the result is large.
which highlights the exploitive nature of its processing of negative decision candidates. In contrast to the slight cost increase in Figure 3.16, all queuing strategies are 40-60% more expensive in the real-world experiment due to non-uniform error distribution. More specifically, we observe for this dataset that 5.2% of all pair-wise decisions are contested, i.e., there exist at least two crowd workers who have a different opinion than the other crowd workers.

Parallelization

Parallelizing the different data interpretation strategies (Figure 3.22) with intra and inter-cluster parallelization methods results in a worse quality/cost development for FEER than FER though both are substantially better than CER. The reason for the efficiency decrease of FEER lies in the noisy information that parallelization induces: It artificially creates race conditions, i.e., edges compete with each other for the highest score which results in them being marked as noisy and removed from the graph. This behavior also explains the high cost of FEER with $\kappa = 0.2$: It generates multiple connections per entity comparison thus further increasing the degree of parallelism and implicitly the noise. Comparing these results to their synthetic counterpart in Figure 3.16(c), we make similar observations for ERS and URS than before while HS increases its advantage over both due to the non-uniform noise distribution in this dataset.
Publications Results

In contrast to the landmarks dataset, the publications dataset consists of records with string-based attributes which allow for an easy application of string similarity metrics to prune the search space and reduce information access cost. The similarity metric $s$ determines the level of matching with the Jaccard similarity as follows: First, if $s$ exceeds an upper threshold, a positive decision with the maximum number of crowd votes is added to the votes graph. Record pairs that are below the lower threshold of $s$ trigger a negative decision in the same manner. Furthermore, $s$ is used to initiate all queues according to the current similarity belief.

Data interpretation

We make three observations when comparing the different data interpretation methodologies (Figure 3.23a and Figure 3.23b). First, fault-tolerant mechanisms clearly outperform consensus-based mechanisms regardless of the similarity threshold $s$ that is applied to the record pairs. Independent of the number of votes spent by Cer, we observe a maximum precision of 0.92 and a maximum recall of 0.29, indicating that, again, Cer keeps records in different entities that should belong to the same entity. The second observation is that spent budget can be significantly reduced if automatic similarity measures are used before applying the data interpretation mechanisms but it does not influence the quality outcome of either technique. This suggests that the decisions made by the similarity metric are in fact decisions that the crowd workers make as well. As a result, tightening the thresholds
only prunes undisputed decisions and thus reduces crowdsourcing costs without modifying the result quality. Third, we observe that for all approaches the cost/quality gain first increases and then stagnates, which is especially evident in Figure 3.23a. The average entity size in the publications dataset is only 3.47 which results in a large number of negative decisions between entity pairs overall. As cost is not bound here, all undetermined pairs have to be resolved until the algorithms are finished. The significant number of negative decisions at the end of the execution (which is visibly more skewed than in the previously examined landmarks dataset) is due to the application of the similarity metric a priori, which boosts decisions that are more likely to be positive to the front of the priority queue.

**Entity connectivity.** The concept of connectivity ratios that was introduced for the landmarks dataset can also be applied to the publications dataset. Similarly to the previous experiment, we observe an increase in cost and quality when $\kappa$ is increased. The changes to the resulting ER are not as significant as observed for the landmarks dataset.

Figure 3.24: Strategy evaluation for Publications dataset (real-world crowd).
but are within 2% of its quality while the cost increases by approximately 10%.

**Summary.** With these experiments, we show that fault-tolerant ER and automatic similarity metrics can be tightly integrated and result in a good quality solution for a smaller overall budget. If the similarity metric suits the dataset, it can efficiently decrease the search space for crowdsourced ER without a loss in quality and an improvement in cost (here 90% of the budget). Furthermore, these experiments show that reducing the search space does not necessarily benefit the decision function, as the output quality is still dependent on how noisy votes are handled. Here, both FER and FEER significantly outperform CER.

**Querying strategies**

With a prepruned search space, querying strategies behave differently than in an unbiased decision space. We observe that for an increased lower bound of $s$, ERS performs better than URS (Figure 3.24b) while URS dominates ERS in an uninformed setup (Figure 3.24a). Obviously, ERS performs well if the candidate space is comprised of mostly positive candidate pairs. On the other hand, it incurs higher cost if it likely asks negative decisions which occur more often if the lower threshold for $s$ is decreased. The development of both HS and URS are more straightforward as they rely on an exploratory way of evaluating the search space which makes them less dependent on prefiltering.

Figure 3.25: Parallel execution for Publications dataset ($0.3 < s < 1$, real-world crowd).
Chapter 3: Data Integration Methodology

Parallelization

Surprisingly, parallelizing the candidate pairs for the publications dataset has significant impact on the performance of Cer (Figure 3.25). Here, the observed f-measure changes because the recall of both Cer variations increases to a maximum of .44 compared to the .29 achieved during sequential execution. The reason for this behavior attests to the instability of consensus-based approaches: It can be found in the order in which the record pairs that are extracted from the priority queue. For this dataset specifically, the ordering of the candidate pairs removes a candidate pair from the initial batch of tasks that has been falsely identified by the crowd and as a result, the output quality improves. In contrast, both Fer and Feer have consistent quality when executed sequentially or in parallel. Figure 3.24c shows the performance of the different querying strategies in parallel execution mode. Similarly to Figure 3.24b, we note that ErS is a valid alternative queuing strategy to HS given a prepruned search space, even if its cost is slightly higher than the cost of HS.

Discussion

Fault-tolerance is a requirement for entity resolution when handling unreliable data sources such as the crowd as shown in this section for two different datasets. Noisy information has direct impact on result quality and cannot be recovered from if the applied ER mechanism is not aware of these imperfections. Furthermore, we show that choices concerning task ordering are essential to the success of any ER mechanism: Queuing strategies as well as parallelization techniques impact the result quality and cost and cause different results in a noisy as in a perfect execution environment. Specifically, we observe that parallelization may improve execution time, i.e., the end-to-end time spent on the ER process, but it increases the allocated budget and often does not achieve the same ER quality level as a sequential execution could provide. The relative success of the presented queuing strategies then is dependent on the dataset itself, the size of the entities it contains, and also whether pre-pruning is an option. We have shown that a hybrid queuing strategy is a robust mechanism to order tasks consciously of the current state of the votes graph as alternative to pure error or uncertainty reduction strategies.

Scalability. For the sake of completeness, these experiments have been executed on datasets that were collected through a crowdsourcing platform. As a result, these datasets

112
are limited in size though experiments with synthetically generated larger datasets show the same tendencies for all algorithms and strategies (see Section 3.1.2.5 for an example of how synthetic experiments are conducted). We argue that as shown in the above real-world experiments, the state-of-the-art mechanisms incur comparable cost in terms of crowd accesses. Thus, fault-tolerant mechanisms on average provide higher quality results at the same cost because they are able to prioritize and question important record pairs that are central to the entity resolution solution. For larger datasets than examined in this set of experiments, we can imagine techniques such as automatic similarity metrics to minimize the candidate space for record pairs similar to those applied on the publications dataset here. Note that this does not diminish the impact of fault-tolerant decision mechanisms on the output quality as shown in our experiments (Section 3.1.2.5).

In terms of computational performance, our experiments show that the entity size as well as noise level in the answer set are correlated to the update propagation performance of \textsc{update} (Algorithm 3). That is, large clusters cause a large amount of positive decisions in the votes graph which in return means more update propagation as positive edges are always traversed independent of whether the candidate path is positive or negative. Furthermore, noisy edges obviously require more computation because every updated edge needs to be propagated into the votes graph. Here, we argue that computational scalability is often not a problem for the execution engine because

- the crowd is slower than the time taken to update the votes graph especially if it is constructed as a chain similar to what all presented approaches here try to achieve by leveraging transitivity.

- parallelization as described previously allows a variety of crowd workers to respond to several tasks at the same time thus decreasing the end-to-end time spent on an ER problem.

(Crowdsourced) Entity Resolution. Automatic ER algorithms and their corresponding approximations, [BBC04, HCML09], are useful alternatives to crowdsourced ER algorithms in a variety of use cases: For example for the Cora dataset which contains text-based content, string-based similarity metrics have been shown to provide high quality output. In this work, the purpose of using this dataset is to be comparable to previously done research in the same area. The use cases that we target with crowdsourced ER for real-world use cases are exemplified better through the landmarks dataset where pair-wise similarities
are not straightforward to compute. In fact, identification of objects in pictures is still a hard task for computers, [WL14]. In these cases, crowdsourced ER can be used to enhance and complement object identification either as a standalone solution or in collaboration with automatic similarity measurements that have been developed for visual computing. Analogous, we imagine crowdsourced ER to be used in other domains where information is not of the same data type or cannot be well correlated with automatic measures.
Algorithm 3. MinMax incremental update algorithm UPDATE and supporting function COMPUTE.

Algorithm: UPDATE($G = (R, E)$, $(r_i, r_j)$) : void

1. compute_new_paths($r_i, r_j$)
   // compute paths of form $r_i - r_j - r_k$
2. $R_i \leftarrow$ COMPUTE($r_j, \{r_i\}, w_{ij} > 0$)
   // compute paths of form $r_j - r_i - r_k$
3. $R_j \leftarrow$ COMPUTE($r_i, \{r_j\}, w_{ij} > 0$)
   // compute paths of form $r_k - r_i - r_j - r_l$
4. foreach $r_k \in R_i$ do
   $R_k \leftarrow$ compute($r_j, \{path_{r_k, r_j}\}, w_{kj} > 0$)
5. resolve($R_i \cup R_j \cup R_k$)
6. RESOLVE($R_i \cup R_j \cup R_k$)
7. Function: COMPUTE($r_i, R_i, \gamma$) : void
8. if $\gamma = false$ then
   // compute negative paths
9. $R_j \leftarrow$ get_records($r_i, R_i$) : $\forall r_j \in R_j : r_j \notin R_i \land p_{ij} > 0$
10. foreach $r_j \in R_j$ do
11. $R_n \leftarrow$ compute_negative_paths($r_i, r_j$)
12. foreach $r_n \in R_n$ do COMPUTE($r_n, R_i \cup r_i, false$)
13. else
   // compute negative and positive paths
14. $R_j \leftarrow$ get_records($r_i, R_j$) : $\forall r_j \in R_j : r_j \notin R_i$
15. foreach $r_j \in R_j$ do
16. $R_n \leftarrow$ compute_new_paths($r_i, r_j$)
17. foreach $r_n \in R_n$ do
18. if $p_{ij} > 0$ then COMPUTE
   ($r_n, R_i \cup r_i, true$)
19. if $n_{ij} > 0$ then COMPUTE
   ($r_n, R_i \cup r_i, false$)
20. return $R_n$
Algorithm 4. Weighted path ER algorithm RESOLVE and supporting function isGood.

1 Algorithm: RESOLVE(G = (R, E)) : ER solution C
2 foreach ri ∈ R; ri /∈ C do
3   ci.add(ri)
4   ci.addAll(rj) : rj ∈ Q; p* ij > n* ij
   // record removal phase
5   foreach rk ∈ ci do
6      if ¬ isGood(rk, ci) then
7         ci.remove(rk)
8      // reset loop in Line 5 and jump there
9      // record addition phase
10     foreach rk ∈ R; rk /∈ ci ∧ ∃ri ∈ ci : p* ik > n* ik do
11        if isGood(rk, ci) then
12           ci.add(rk)
13      // reset loop in Line 9 and jump there
14     if ci = ∅ then
15        ci.add(ri)
16     C.add(ci)
17 return C;

18 Function: isGood(ri, ci) : boolean
19 score, penalty = 0
20 foreach rj ∈ ci; ri ≠ rj do
21      if p* kl > n* kl then score += p* kl - n* kl
22     else penalty += n* kl - p* kl
23 return score>penalty
3.2 Data Cleaning

We now discuss the second data integration task that we examined in our work, namely data cleaning. This part of this dissertation is also discussed in [GHG16]. Data cleaning is an important research area because of the ever-increasing amounts of data that heterogeneous information systems handle. Word ambiguity, misspellings, or missing data are just a few examples for what is called *dirty data* and what causes data analysts to spend up to 80% of their time cleaning data instead of analyzing it [KPHH12]. Systems such as Tamr [SBI+13], Wrangler [KPHH11], or Wisteria [HKW+15] address this problem and (help to) establish interactive cleaning pipelines that leverage user knowledge to improve their data. Supporting semi-automatic data repair systems, research work in this area has also focused on automatic data cleaning. Automatic cleaning techniques often leverage the semantic context of data values: if records in a data table show similar behavior across columns, they are less likely to be false. As an example, take the data cleaning example mentioned earlier in the introduction, Figure 1.3. In this example, there exists a clear semantic relationship (a functional dependency) between the two columns, *City* and *Country*. Additionally, five of the six values show the same standardized characteristics, i.e., a 3-letter upper case code, while *r*$_5$ has a different appearance. In this example exist two types of errors. First, *r*$_1$ contains a syntactic error that may have been caused by mistyping the abbreviation. Second, *r*$_5$ contains a semantic error that although fitting the topic of the column does not fit the remaining values in the column *Country*. Techniques such as [BFG+07] leverage functional dependencies (FDs) to determine erroneous values. However, their drawback is that they depend on this kind of contextual information (a second column) to find them. Instead, we observe that even within a column correlation exists [CHCG15] and propose to leverage these to identify outliers independent of the column’s context. As a result, our technique is applicable in a broad variety of use cases. For the running example specifically, we are able to detect the semantic dependencies in column *Country* without the knowledge of column *City* which is otherwise only possible through manual repairs [CM08]. This makes our techniques especially useful for on-the-fly corrections of enumerations, spreadsheets, etc. but can also be used to enhance traditional FD-based approaches.

Going back to the example shown in Figure 1.3, our work proposes an automatic cleaning mechanism that identifies outliers in an input set, here column *Country*. For that, we first explore how correlation between data items, also referred to as *tokens*, can be captured...
accurately. Given a tabular data structure, a token represents a cell entry in a data column. Thus, a token does not necessarily refer to a single word but can also be a group of words, for example if the observed column contains names and one of these is ‘Bill Gates’, we use these two words as one token. Correlation between tokens is captured by clustering the tokens into semantically consistent groups. To find these groups, we leverage tabular data such as found on the web or within an enterprise. Tabular data often has high consistency as shown in [CHW+08] and can therefore be used as a to identify how to correlate a token with other tokens in the input set. Using the knowledge of these correlations, we can then establish the semantically consistent groups within the input data and mark those (groups of) tokens that are considered outliers.

Our work is based on the key observation that tokens occurring in the same column in a table have a semantic relationship, i.e., that they point to the same underlying semantic group and can be thus clustered together. To construct these groups, we leverage the co-occurrence of tokens through online group similarity computation. In comparison to naive techniques that employ exhaustive pair-wise computation metrics, group similarity can be computed in linear time and is thus applicable not only for static cleaning processes but can be employed for online data cleaning. To compute the cohesiveness of groups of tokens, we use and extend a metric called specific correlation [Van11]. We then couple this metric with agglomerative clustering techniques that construct the group clustering for a set of tokens to determine outliers or groups thereof. Finally, we outline how correlation between tokens can be leveraged to correct outliers.

To the best of our knowledge, our work is the first to use single-column semantic correlation of tokens to propose data cleaning opportunities. We make the following contributions:

- **Group Similarity Computation.** We show how correlation between tokens can be accurately captured in a framework that can be used for efficient similarity computation. Furthermore, we introduce the notion of group similarity to compute the similarity between sets of tokens rather than pair-wise similarities between tokens.

- **Outlier Detection.** We develop agglomerative clustering techniques that leverage token correlation to construct semantically consistent cluster. This technique is then leveraged to determine outliers in token sets.
3.2: Data Cleaning

3.2.1 Problem Overview

In this work, we discuss a general approach to outlier detection using a corpus of previously observed sets of related tokens. The input of our system is two-fold. First, we collect and process data to observe the co-occurrence of tokens to build a correlation graph. This data can be obtained from the web (web tables such as found on Wikipedia) or could be specific to an enterprise (for example derived from spreadsheets). The construction of the correlation graph is done offline or as a background process that occasionally updates the token co-occurrence information. The second part of our work focuses on the actual outlier detection for a set of tokens in near real-time, i.e., as an online process.

In Section 1.1.1.2 we shortly outlined the problem of semantic outlier detection which is to find those tokens in an input set that do not fit the same semantic context as the other tokens. We now reformulate this problem as a more explicit clustering problem which applies the terminology used throughout this work. Let \( R \) be the tokens that are the input to the system, i.e., the column or set of tokens that the user wants to confirm as being semantically consistent. For each token \( r_i \in R \), the system assigns those \( r_i, \ldots, r_n \) to a cluster \( c_j \) that are empirically shown to be similar and thus likely belong to the same semantically consistent group. For example the semantic group that most values in the running example belong to is a three-digit country code. During the execution of the outlier detection algorithm, we do not specify which group a set of tokens belongs to. Instead, we only identify whether (a subset of) tokens belong (not) to the same group to identify outliers. This is expressed in a clustering \( C \) where each \( c_j \in C \) is associated with a semantically consistent group of the tokens \( r_i \) that belong to \( c_j \). Note that determining the semantic correlation for \( r_i \) is not trivial as some tokens may be conceptually associated with multiple groups. For example, the ISO 3166 three digit code for Monaco in the introductory example (Figure 1.3) is MON which is simultaneously the abbreviation for the weekday Monday.

**Problem 3.2.1 (Semantic Clustering).** Semantic clustering is the task of finding a clustering \( C = \{c_1, c_2, \ldots, c_n\} \) for an input set of tokens \( R \), where the clustering is disjoint with \( c_i \cap c_j = \emptyset, \forall i \neq j \) and \( \bigcup_{i \in [n]} c_i = R \), such that every cluster \( c_i \in C \) points to a different semantic group.

In the introductory example, \( C \) consists of one cluster \( c_1 \) with four records \{USA, GER, CAN, MON\}, a cluster \( c_2 \) with one record \{Switzerland\}, and a cluster \( c_3 \) with record
{ATU}. That means that for each of these three clusters, the consistency within the existing cluster is higher than merging it with any of the other two clusters. To compute the consistency of a cluster, we introduce the notion of group similarity. Instead of computing pair-wise similarities between all tokens in the input set and then determining a clustering on top of the resulting graph, the idea of group consistency is to directly calculate the collective similarity of a set of tokens. Group similarity is formally defined in Section 3.2.2.

As mentioned previously, our outlier detection technique addresses two types of outliers. First, we want to discover syntactic outliers, i.e., tokens such as ATU in the running example, which are often caused by mistyping the actual token. The second type of outlier that the system identifies are semantic outliers such as Switzerland. Obviously, identifying these outliers is only the first step in the process of data cleaning. This work therefore discusses further how the correlation graph can help to propose corrections for both types of outliers. For example, one transformation that the system should be able to identify is the syntactic transformation of ATU to AUT which is the correct country code for ‘Austria’. This can be achieved by a) finding those tokens that have a high correlation to those tokens in \( c_1 \) and b) applying a syntactic similarity function to determine the similarity between the candidate repairs with the outlier. We discuss the notion of correcting tokens in Section 3.2.4.

**Framework Description**

To enable outlier detection as well as corrections, the system first needs to determine a token’s possibly associated semantic groups in what we refer to as the outlier identification step. In our work, we do not precompute these groups but rather use the data collected from different data sources such as web tables offline to interpret the token dependencies it indicates in an online process.

To capture web or enterprise data accurately, we construct a bipartite graph \( G = (U, V, E) \) that documents the relationship between tokens and the context that they were observed in. This graph serves as our correlation graph. The two partitions of the graph are tokens (\( U \)) and column identifiers (or sources) (\( V \)) as obtained from structured data such as web tables. There exists an edge connecting a token \( u_i \in U \) with a source \( v_j \in V \) if the token appears in \( v_j \). If there exist two tokens \( u_i \) and \( u_k \) that have an edge to \( v_j \) we say that this is evidence that they belong to the same semantic group. Take as an example the bipartite graph shown in Figure 3.26. This graph is computed by examining the columns observed.
in a web corpus and constructing the corresponding token-source graph. Multiple edges of the same tokens pointing to the same data source hint that these tokens point to the same underlying group because they frequently co-occur in their reference columns. For example tokens USA and GER appear in columns Abbreviations and A3 while Monaco and GER never co-occur. This indicates that Monaco and GER belong to different semantic groups. Using the correlation graph, we are able to compute the similarity of (a set of) tokens. At the same time, it allows us to say which other tokens co-occur frequently given a (set of) token(s). These two traits are leveraged for outlier detection and token corrections as discussed in the next sections.
Chapter 3: Data Integration Methodology

3.2.2 Outlier Identification

What we describe as outlier identification in our work is related to research in clustering which is often associated with deduplication and classification in the context of data integration [HCML09, MCM13]. All of these techniques have in common that they want to find one or multiple clusters for a set of tokens. In deduplication, these clusters represent for example the same real-world entities which are found through pair-wise similarity computation. Similarly, machine learning clustering algorithms obtain different classes for the data items by finding dense clusters (where each cluster represents a class) [HA04]. In our work, we build a framework that allows us to identify semantically consistent groups of tokens which we then leverage for outlier detection and token correction.

Bipartite Relationship Graph

We base our outlier computation on a bipartite token source (TS) graph that captures the relationships between tokens $u_i$ and data sources $v_j$ such as columns in web tables. We choose this data storage for two reasons.

Similarity Computation Flexibility. The similarity computation that we describe in Section 3.2.3 uses the notion of group similarity, i.e., how similar a (variably sized) set of tokens is. To compute this similarity, we do not apply a static clustering technique but compute similarities on the fly. This methodology requires a flexible data structure that allows us to compute the similarity of pairs, triples, quadruples, and so on without causing additional storage overhead. By storing the dependencies of a single token, we can dynamically select which tokens are relevant for the current input and compute their similarity without restrictions.

Suggestion Inference. To obtain candidates to correct a token with, we require a data structure that helps us to explore candidates that belong to the same semantic group as the reference tokens and syntactically or semantically similar to the outliers. Using the TS graph, we can intuitively exploit its bidirectional edges for that purpose. For example if we want to find tokens that are indirectly connected to $MON$ in Figure 3.27, we first find those columns in which $MON$ can be observed. These columns are $A3$ and $Days$. In the second step, we examine the syntactic or semantic relationships of the tokens to be corrected with these candidates.
The TS graph is in essence an occurrence graph (OI) which maps the occurrences of tokens to their corresponding sources across various extracted tables. As an alternative to the TS graph, we explored graph storage models that store higher-level similarities for all tokens in memory instead of computing them on the fly as required when the OI graph is used for computing the similarity of groups. It is an interesting alternative because a) a lot of the clustering algorithms that can be applied for this problem (for example correlation clustering [BBC04] or average link [CPKT92]) leverage co-occurrence metrics and b) it minimizes the computation overhead during the online execution. A major drawback and the reason why we decided to compute these similarities online is storage space. Figure 3.28 shows the storage requirements for a co-occurrence index (COI) that stores all pair-wise similarities and the bipartite occurrence index that was described previously as our knowledge base of choice. The COI basically precomputes all pair-wise similarities and stores those in a hash index. Thus, access time for the COI is constant. However, the storage cost for COI grows quadratically while the cost for the OI grows linearly. Obviously, storing co-occurrence information of even higher orders exponentially increases the storage space. Furthermore, using an OI as storage structure has another advantage compared to the COI which refers back to the first key requirement for our relationship graph: It provides means for a flexible similarity computation for an arbitrary group of values. Recall that the COI stores pair-wise similarities only. In contrast, the OI enables us to compute k-similarities, i.e., for triples, quadruples etc., on the fly. How this property can be leveraged for outlier identification is described next.
Chapter 3: Data Integration Methodology

Similarity Computation

The term group similarity describes the semantic relationships of a set of tokens. In this work, we use group similarity to estimate the goodness of a set of tokens given knowledge of how often these tokens co-occur. In contrast, prior work has measured group (or cluster) correlation mostly through pair-wise comparisons of tokens. For example, we want to establish whether tokens GER, USA, CAN, and MON belong to the same semantic group. Using pair-wise similarity, we could compute the similarity between each pair of tokens using pointwise mutual information (PMI) \[ \text{PMI} \] as shown in Figure 3.29 (a). The similarity of tokens within the group is then the average (or minimum/maximum, depending on the application) PMI of these token similarities. The PMI is formally defined as:

\[
\text{PMI}(r_i; r_j) = \log \frac{p(r_i, r_j)}{p(r_i)p(r_j)}
\] (3.3)

To compare the PMI of several token pairs, we use the normalized PMI (NPMI) \[ \text{NPMI} \].

\[
\text{NPMI}(r_i; r_j) = \frac{\text{PMI}(r_i; r_j)}{-\log(p(r_i, r_j))}
\] (3.4)

Basically, the NPMI maps the similarity of two records \( r_i \) and \( r_j \) to \([-1,1]\). If NPMI\((r_i;r_j)=1\) then \( r_i \) appears only in conjunction with \( r_j \) and vice versa. If NPMI\((r_i;r_j)=-1\) then \( r_i \) and \( r_j \) can never be found in the same column. Finally, the NPMI accounts for randomness, i.e., if NPMI\((r_i;r_j)=0\) then \( r_i \) and \( r_j \) appear independently of each other but there may exist co-occurrences of both values. The likelihood of \( r_i \) and \( r_j \) appearing in the same column, \( p(r_i, r_j) \), is computed by intersecting the columns in which \( r_i \) and \( r_j \) occur and
normalizing the score with respect to the total corpus. If $V_i$ are those columns that $r_i$ maps to and $V_j$ the column ids corresponding to $r_j$ then $p(r_i, r_j) = \frac{|V_i \cap V_j|}{|V|}$.

Example 3.2.1. In Figure 3.29 (a) the NPMI leads to similarities that all are above 0, i.e., the tokens are all somewhat correlated, but we see differences in the co-occurrence scores of values. Specifically, we observe that MON is not as well-connected as all other three values: It is connected to CAN with $\text{NPMI}(\text{MON, CAN}) = 0.46$ which is the best score connecting MON to any other token while its score with token USA is only 0.17.

The reason for the discrepancy observed in the example is that token MON is ambiguous: It means ‘Monaco’ in the standardized ISO code but it may also stand for ‘Monday’ if used as an abbreviation for the weekday. Clustering records based on pair-wise similarities first of all requires the algorithms to know the pair-wise similarities for all possible combinations in the input set. Algorithms such as the aforementioned correlation clustering (CC) or Avg-Link which averages the pair-wise similarity of a cluster as its density value are then applied on top of the resulting similarity graph. In practice, this may not necessarily lead to a good result as shown in Figure 3.29 (a) where CC splits the cluster under the assumption of a clustering threshold of 0.4. At the same time, computing all pair-wise similarities is costly as executing any of these algorithms is in $O(n^2)$ where $n$ is the number of nodes in the input set. To address this problem, we next discuss a novel clustering metric called normalized specific correlation which estimates the similarity of groups instead of pairs and can be computed in linear time.

Group Similarity. To compute group similarity, we use an extended version of PMI called specific correlation [Van11]. It is formally defined as follows.

$$SC(r_1; \ldots; r_n) = \log \frac{p(r_1, \ldots, r_n)}{\Pi_{i=1}^n p(r_i)}$$ (3.5)
It characterizes the similarity score of a set of tokens through the ratio of their co-
ocurrence likelihood \( p(r_1,\ldots,r_n) \) to the occurrences of each token separately. Analogous to the PMI metric, we need to normalize its score to obtain comparable results for different token set sizes. Therefore, we introduce a novel metric called \textit{normalized SC (NSC)}. This metric computes the SC score in relation to the number of tokens in the token set and their group co-occurrence.

\[
NSC(r_1;\ldots;r_n) = \frac{SC(r_1;\ldots;r_n)}{(1-n) \log(p(r_1,\ldots,r_n))} 
\]  

(3.6)

We note that our new metric NSC properly normalizes SC for positive value co-occurrence into the range of \([0,1]\), analogous to the commonly used NPMI normalization of PMI.

**Property 3.2.1** (Proper Normalization). \textit{NSC normalizes SC properly and in a way similar to the NPMI normalization. Specifically, NSC = 1 if all values have complete co-occurrence, and NSC = 0 if they are completely independent.}

\textit{Proof.} We first show that NSC = 1 if all values are completely co-occurring. In that case \( p(r_1,\ldots,r_n) = p(r_1) = \ldots = p(r_n) \) holds. Let \( c \) be the value of these (equivalent) probabilities. Then we have

\[
SC = \log(p(r_1,\ldots,r_n)) - \Sigma_{i=1}^n \log(p(r_i)) 
= \log(c) - \Sigma_{i=1}^n \log(c) 
\]  

(3.7)

(3.8)

Normalizing SC using Equation (3.6), we have

\[
NSC = \frac{\log(c) - \Sigma_{i=1}^n \log(c)}{(1-n) \log(c)} = 1 
\]  

(3.9)

Next, we show that NSC = 0 if all values are independent. This is easy to see, because if all values are independent, we have \( \prod_{i=1}^n p(r_i) = 1 \), thus SC = 0 and also NSC = 0.

\[\square\]

In this context, we only focus in positive co-occurrence (i.e., the range \([0,1]\) for NSC), as negative co-occurrence is not relevant for the purpose of clustering semantic groups. Furthermore, we make the following observation about the relationship of the occurrence of a token and its co-occurrence with other tokens.

**Property 3.2.2** (Interdependence of Probabilities). \textit{The dependency between the co-occurrence likelihood \( p(r_1,\ldots,r_n) \) and the occurrence of each token \( r_i \) with fixed NSC score...
NSC is computed as
\[ p(r_1, \ldots, r_n) = e^x, \text{ with } x = \frac{\sum_i \ln(p_i)}{1 - (1 - n)NSC} \] (3.10)

Thus, an increase in any \( p_i \) means that \( p(r_1, \ldots, r_n) \) has to increase as well to maintain the NSC score.

We observe that the NSC metric ensures that if a token occurs less often in the dataset, its co-occurrence with the other tokens can be relatively smaller to achieve the same NSC score. This ensures that rare occurrences of a token are weighed appropriately when considering the mapping to an underlying semantic group. Furthermore, it allows us to make the following observation of the dependency of the co-occurrence likelihood with the group size \( n \):

Lemma 3.2.1 (Variation of Group Size). Increasing the number of tokens in the token set by one token \( r_k \) decreases the required co-occurrence likelihood for obtaining the same NSC score if for the occurrence score \( p(r_k) \) holds that:

\[ p(r_k) < e^y \text{ with } y = \frac{(\sum_i \ln(p_i)) * NSC}{1 - (1 - n)NSC} \] (3.11)

If \( p(r_k) = e^y \), then the co-occurrence count remains the same to achieve NSC, otherwise it increases.

Proof. This lemma follows directly from Equation (3.10). The same co-occurrence count is obtained if

\[ \sum_j \ln(p_j) = \frac{(\sum_i \ln(p_i)) * (1 - (1 - (n + 1))NSC)}{1 - (1 - n)NSC} \] (3.12)

holds. In words, \( x \) in Equation (3.10) has to be equivalent for both token set variations with \( \sum_j \ln(p_j) \) denoting the co-occurrence likelihood of \( r_1, \ldots, r_n, r_k \). Resolving Equation (3.12) to \( p(r_k) \) gives the above threshold for the occurrence probability of \( r_k \).

At the same time, this shows that for each addition of new token, we incur an increasingly high occurrence count to maintain the same NSC score. However, if the \( p(r_k) \) remains constant and is not adjusted, the NSC will decrease as the co-occurrence score decreases. This is visualized in Figure 3.30. Here, every token of a group with size \( n \) has an occurrence likelihood of \( p(r_i) = 0.05 \) and \( n \) is varied. We observe that the break-even point with
Chapter 3: Data Integration Methodology

Figure 3.30: NSC score development for varying group with $p(r_1)=0.05$.

$NSC = 0.5$ is reached at a lower value for $p(r_1, \ldots, r_n)$ with $n = 4$ than $n = 3$ which is exactly the behavior we expect given the above properties of $NSC$ with $p(r_i) < p(r_k)$. Intuitively, this behavior also means that a group with more tokens has a higher consistency than its subset if the co-occurrence counts of both are the same. For example, if the four tokens $USA$, $MON$, $GER$, and $CAN$ co-occur 198 times, the $NSC$ score is higher than if only $USA$, $MON$, and $GER$ co-occur 198 times. That means that larger token sets are rewarded for co-occurrence more substantially than smaller groups.

**Example 3.2.2.** In general, not every token occurs equivalently. Our running example is taken from a web corpus dataset in which $USA$ appears in approx. 188K different columns and $MON$ only appears in about 7.2K different columns. They co-occur in 495 different columns which is a relatively small number and the reason why the $NSC$ of this pair is only 0.17. If we extend the pair to ($USA$, $MON$, $GER$) we observe 242 co-occurrences and adding $CAN$ leads to a reduction to 198 column ids. The absolute reduction of column ids is not as high as expected (i.e., as if they were uncorrelated) which leads to an increase of the $NSC$ from 0.17 to 0.33 and finally 0.41 (see Figure 3.29) which signals an increasing level of consistency.

**Theorem 3.2.1** (Complexity of NSC). The $NSC$ score can be computed in $O(n)$.

The linear computation time of the $NSC$ score follows directly from its definition. To compute the score, we determine the intersection of the token’s source mapping in addition to the corresponding $p(r_i)$ scores. As this can be done with one look-up per token using the bipartite relationship graph, the $NSC$ computation can be done in linear time.
3.2.3 Outlier Detection

As explained previously, semantic clustering when applied in the context of outlier detection is the task of finding the correlation between a set of input tokens $R$ and determining which of these belong to the same semantic group. One special case of semantic clustering is clustering under the assumption that there exists exactly one underlying semantic group of tokens that each token in the input set should belong to. Thus, every token that does not belong to that semantic group is an outlier. The clustering $C$ then contains one cluster $c_i$ with tokens representative of that group and a set of single-token clusters that represent the outliers. This clustering variation is usually applied for applications such as online spreadsheets or tabular content which are edited by users and assume the user’s consistency. Another use case for outlier detection to find multiple semantic groups in $R$.

As an example take a data integration scenario where one data source uses the full country names as standardization and the second data source its abbreviations. Merged together, the input would be inconsistent which should be identified to make the data consistent for further processing.

To address both of these outlier detection scenarios, we will describe outlier detection based on the single-group assumption first. Afterwards, we will extend it to address the multi-group problem.

Optimal Algorithm

An optimal single-group outlier detection solution consists of one cluster $c_i$ with multiple records that have the same underlying group and potentially multiple singleton clusters, i.e., clusters that contain only one record. In this case, $c_i$ contains tokens that adhere to the dominant semantic group for the input tokens while the singleton clusters contain outliers to that group. To differentiate between multiple candidate clusterings that fulfill this structural condition, we term the best clustering the one with the largest consistent cluster and the highest NSC score. A consistent cluster in this context refers to a cluster $c_i$ for which holds that any cluster $c_j \subset c_i$ does not have a higher NSC than $c_i$.

**Definition 3.2.1 (Consistent Clusters).** Let $c_i$ be a cluster in a clustering $C$. We say that $c_i$ is consistent if there exists no $c_j \subset c_i$ for which $\text{NSC}(c_j) > \text{NSC}(c_i)$.

Consistency is a requirement for optimal semantic clustering because a lower NSC for a subcluster means a diffusion of the cluster density as discussed in Section 3.2.2.
Definition 3.2.2 (Optimal Single-Group Clustering). Let $C$ be a clustering with one consistent cluster $c_i \in C$ that contains $k \geq 1$ records and $n - k$ singleton clusters $c_j$. For $C$ to become the best clustering $C^*$, there cannot exist an alternative clustering $C'$ for which $|c'_i| > |c_i|$ or $\text{NSC}(c'_i) > \text{NSC}(c_i)$ holds.

In other words, we choose the clustering alternative $C$ as $C^*$ for which $c_i$ is maximal, i.e., there exists no other $c'_i$ that contains more tokens or that has a higher score with the same number of tokens while being consistent.

Multiple Semantic Groups. Analogous to single-group clustering, consistency is a key requirement for multi-group clustering as it ensures that the clusters are semantically cohesive. To determine valid solutions for the multi-group clustering problem, we extend the notion of optimal single-group clustering as follows: $C$ may contain multiple clusters $c_i$ for which $|c_i| > 1$ and the consistency constraint described in Definition 3.2.1 holds. In practice, this definition of an optimal multi-group clustering may again lead to multiple candidate clusterings. To decide which clustering is then preferred, different decision mechanisms can be employed. For example, one approach is to determine that a clustering $C$ is $C^*$ if it is optimal and its biggest cluster $c_i \in C$ is bigger than any cluster $c'_i$ in any of the other candidate clusterings. This way of choosing $C^*$ is in line with the idea of outlier detection presented previously. Another solution would be to maximize the number of tokens in non-singleton clusters. Which approach is chosen, depends on the goal of the multi-group clustering process.

Greedy Optimal Algorithm. Finding $C^*$ is difficult to compute in practice as of now because the system would need to generate all permutations of candidate clusterings, evaluate their consistency, and finally order them according to their cluster sizes to verify all solutions. To approximate $C^*$, we now introduce a greedy version for the optimal clustering problem which is a variant of a bottom-up agglomerative clustering technique also referred to as hierarchical clustering [JD88]. Algorithm 5 describes GREEDYOPT, a greedy algorithm that solves optimal multi-group clustering, in pseudo code. The algorithm takes as input the input set $R$ as well as a TS graph $G = (U, V, E)$. It initializes the output of the semantic clustering $C$ with singleton clusters, i.e., every token is assigned its own cluster. It then proceeds iteratively: First, the algorithm picks a cluster $c_i$ at random. It then examines all other remaining clusters $c_j$ in the queue and computes their NSC scores together with $c_i$. If the combined NSC score of $c_i$ and $c_j$ exceeds their single scores, then we have found a cluster that is more consistent together than apart (Line 6). Thus, $c_i$ and
3.2: Data Cleaning

**Algorithm 5.** Greedy agglomerative semantic clustering algorithm `GREEDYOPT`.

**Input:** Repair Tokens $R$; TS Graph $G = (U, V, E)$

**Output:** Clustering $C$

1. $C \leftarrow \text{CreateSingletonClusters}(R)$
2. $Q \leftarrow C.\text{ClusterIDs}$
3. while $\neg Q.\text{empty}$ do
   4. $c_i \leftarrow Q.\text{PollRandom()}$
   5. foreach $c_j \in Q$ do
      6. if $\text{NSC}(c_i \cup c_j) > \max[\text{NSC}(c_i), \text{NSC}(c_j)]$ then
         7. $C.\text{Merge}(c_i, c_j)$
         8. $Q.\text{ResetTo}(C.\text{ClusterIDs})$
         9. Jump to Line 3
6. return $C$

$c_j$ are merged and the queue is reset (Lines 7 and 8) after which the algorithm jumps back to the beginning of the loop (Line 3). Note that this algorithm is designed for multi-group clustering. However, it can be easily transformed to outlier detection by modifying Line 4 to pick the largest cluster instead of a random cluster thus maintaining one big cluster representing the corresponding semantic group while outliers remain in singleton clusters.

**Approximate Algorithm**

Even though `GREEDYOPT` algorithm is a good approximation for finding the optimal outlier detection problem, it is not well applicable for real-world datasets because these datasets are noisy. Recall the condition under which we consider a merge of two (singleton) clusters: their combined NSC has to outweigh the individual NSC scores. Monotonicity is only guaranteed if adding any new token to the group strengthens the group which is often not the case in datasets with errors. Amongst others, tokens such as `MON` can point to multiple semantic groups. As a result, `GREEDYOPT` may fall into local optima which it cannot recover from. Most similarity-based algorithms therefore use the notion of a similarity threshold $\alpha$ instead of the consistency requirement defined above. Tokens are considered to belong to the same group if $\alpha$ is exceeded.
Chapter 3: Data Integration Methodology

Definition 3.2.3 (Approx. Single-Group Clustering). The best approximate clustering $C^*$ contains a cluster $c_i \in C^*$ for which NSC($c_i$) exceeds $\alpha$ and $c_i$ is maximal, i.e., there exists no $C'$ for which $|c'_i| > |c_i|$ holds.

Multiple Semantic Groups. Analogous to optimal multi-group clustering, approximate multi-group clustering extends the above definition by allowing multiple multi-token sets to exist in the final clustering solution.

Greedy Approximate Algorithm. Similar to finding the optimal clustering, exhaustive approximate clustering examines all permutations of clusters to find the best one according to Definition 3.2.3. As the computational effort for large input sets is high, we introduce a greedy version to solve this problem analogous to the previously presented GreedyOpt algorithm. For the GreedyApprox algorithm, we only have to modify Line 6 in Algorithm 5 such that if $\text{NSC}(c_i \cup c_j) > \alpha$ holds, the merge of $c_i$ and $c_j$ is triggered. This approach is very efficient but is prone to local optima because of its merge decision order. To resolve situations where clusters are merged that barely exceed $\alpha$ first, we introduce a second approximation GreedyApproxTight that builds upon GreedyApprox. It is a two-threshold version of GreedyApprox and prioritizes merge decisions for which $\text{NSC}(c_i \cup c_j) > \frac{1-\alpha}{2} + \alpha$ holds. In other words, it artificially tightens the acceptance threshold when it decides whether to merge two clusters. If no merge decision is found for this tighter threshold, it falls back on the decision function of GreedyApprox. While evaluation our algorithms, we have experimented with several approaches that tighten the threshold or reorder the merge choices according to the NSC scores. In practice, we find that GreedyApproxTight provides better performance than sorting algorithms while providing similar result quality. Furthermore, the choice of any (tightened) threshold is a general concern for a threshold-based semantic clustering. In our work, we use a sample of the dataset and vary the threshold to find a suitable $\alpha$ for all the approaches that we tested. Details about how thresholds change for two different datasets and even between different threshold-based algorithms are described in Section 3.2.5.

Again, note that this methodology describes multi-group clustering. It can be adapted to outlier detection in a straightforward manner analogous to Section 3.2.3.
3.2.4 Token Corrections

Correcting a token as described previously addresses two error types. First, *syntactic* errors are those errors that occur when misspelling a token. Second, *semantic* errors are formatting errors that most commonly occur when data is integrated from multiple data sources through schema matching. Both of these are marked in our semantic clustering step as outliers. For cleaning syntactically erroneous tokens, we propose to leverage the token-source relationships in the TS graph and similarity computation in two steps, *suggestion generation* and *scoring*.

**Suggestion Generation.** Extending the neighborhood of the tokens in the reference cluster, we determine those tokens that are potential corrections for the outlier.

**Scoring.** If there are multiple candidate repairs, we compute the similarity between the original and the new token as well as the consistency of the new token with the reference cluster. The resulting score is the likelihood of the new token being a repair candidate.

Addressing semantic errors is not as straightforward as fixing syntactic errors. We will discuss the challenges for this problem as well as some initial ideas on how to solve it at the end of this section.

**Suggestion Generation**

To generate candidate tokens, we first exploit the characteristics of the TS graph. As shown in Section 3.2.2, the bidirectional edges between tokens and observed sets, or columns, in the bipartite graph can be used to find the neighborhood of tokens in a cluster. Given a token $r_o$ that has been identified as an outlier and a reference cluster $c_i$, we expand the group of $c_i$ in the first step of our algorithm for suggestion generation. That is, we first find the 1-hop neighborhood $V_k \subseteq V$ of those semantic groups that contain all values $r_j \in c_i$. In the second step we then use the token mapping functionality of the columns and retrieve the candidate values $U_l \subseteq U$ that are referenced by any of the column values in $V_k$. Any token $u_n \in U_l$ is a token that could be good correction of our outlier. However, as there exist a variety of co-occurrences of tokens, the search space albeit reduced is still relatively large at this point. We thus reduce it further automatically as described in the next step, scoring.
Scoring

The scoring of candidate tokens is workload dependent. For example, if we know that the we expect syntactic errors as outliers in our dataset, we handle them differently than more consistently occurring semantic errors. As a result, we now sketch a scoring mechanism that is applicable for both scenarios. To determine which candidate is a good correction for the token $r_o$, we compute the similarity of $u_n$ with $r_o$ as well as the group consistency $g$ of $u_n$ with respect to $c_i$ as $score(r_o, u_n, c_i)$. Similarity computation for syntactic errors commonly uses metrics such as the Levenshtein edit distance or the Jaro-Winkler distance [CRF03]. The decision which metric to use is dependent on the workload. Using group similarity on the other hand ensures that $u_n$ fits the logical semantic group that $c_i$ is based on. The score can then be computed as follows.

$$score(r_o, u_n, c_i) = w_s \times sim(r_o, u_n) + w_d \times g(u_n, c_i)$$

In this equation, $w_s$ and $w_d$ represent weights that determine the importance of each of the components. As mentioned above, these can then be adapted to the varying scenarios where token correction is applied.

**Example 3.2.3 (Scoring).** Given are clusters $c_2$ with value Switzerland and $c_3$ with syntactically erroneous token ATU as well as the reference cluster $c_1$ with $\{\text{GER, USA, AUS, MON}\}$ from the running example. Extending the neighborhood returns values SUI and AUT that are indirectly connected through column ‘A3’ as visualized in Figure 3.31. To determine the syntactic correction for ATU, we compute the Jaro-Winkler similarity for
(ATU, SUI) and (ATU, AUT) which is 0.56 and 0.9 respectively. Thus, the score for these candidate tokens is \( \text{score}_{\text{SUI}}(\text{ATU}) = 0.505 \) and \( \text{score}_{\text{AUT}}(\text{ATU}) = 0.675 \). Here, AUT is the preferred syntactic correction for ATU.

Repairing semantic errors is not as straightforward because usually, the string similarity metrics cannot be applied as easily. For example in Figure 1.3, we observed that in order to fix token ‘Switzerland’, we would have to apply a transformation mechanism that identifies its match with ‘SUI’. However, syntactically speaking, these tokens are very different. Thus, we discuss how we can derive good similarity functions in scenarios like this one next.

Discussion

Finding a good, generally applicable, similarity function is an open research problem. As we describe above, the current standard is to fit similarity functions to the workload. We argue that given the outlier detection mechanisms that we have introduced, our research can be leveraged to help finding these similarity functions. The output of the semantic clustering phase are a set of clusters that each are semantically consistent. Thus, let a transformation function \( T \) describe the mapping of values in one cluster to another cluster. We make two observations that help to determine a good function \( T \) that is applicable for the respective clustering context.

1. If the clustering contains multiple large clusters, this is an indication that specific characteristics of each cluster can be leveraged in the transformation function.

2. If several small clusters are part of the clustering, they indicate unique errors such as could be the case when making syntactic mistakes.

Again, finding this transformation function is a hard problem and one that we do not address in this work. However, we argue that these observations about the dependencies within a semantic clustering are a good starting point for future research on transformation functions. Furthermore, in order to reduce the search space for these transformations, operations such as outlier detection are crucial which already give hints as to which possible correction methodology may be required.
3.2.5 Experimental Evaluation

In this section, we discuss our experimental results for the different outlier detection algorithms and show some initial potential applications for token corrections. We use two different datasets in this evaluation, described in detail next.

**Enterprise Dataset.** This dataset has been extracted from a real-world enterprise and consists of 9,485 token sets with 81,243 tokens. Each token set corresponds to one column in an enterprise spreadsheet. The corresponding bipartite graph has a size of 5.5MB. To provide an accurate ground truth, we randomly picked 100 columns from these and manually constructed their ground truth.

**Web Dataset.** The web dataset is a collection of 15,915,233 token sets containing 92,007 tokens collected by crawling web tables. The token sets in this dataset correspond to columns that have been found in web tables. The 1.4GB bipartite graph for this dataset is based on the parsed web data and is relatively noisy, i.e., tokens often refer to multiple domains. The ground truth for this dataset is derived by parsing Wikipedia and is kept separate from the columnar data used to build the bipartite graph. We use Wikipedia as ground truth because it has high average accuracy (its tabular data is controlled and manually assembled by experts).

To test these datasets, we implemented the following outlier detection algorithms:

**GreedyOpt (GO).** Agglomerative clustering technique that iteratively merges clusters bottom-up if the newly generated cluster has a higher consistency, i.e., a higher NSC score, than either of the previous clusters.

**GreedyApprox (GA).** Bottom-Up agglomerative clustering technique that merges two clusters if their combined NSC score exceeds a similarity threshold $\alpha$.

**GreedyApproxTight (GAT).** Extension of GA that first tightens the threshold to find more consistent merge options. The idea here is that the higher the NSC score, the more relevant a merge decision. GAT therefore prioritizes highly scored decisions before falling back on lower (but still above the threshold) decisions.
3.2: Data Cleaning

**Exhaustive Approx (EA).** This technique is equivalent to GA but instead of choosing the first possible merge option, it chooses the best possible operation per iteration in the agglomerative clustering.

All of these clustering algorithms use the *NSC* score for similarity computation. We further provide two baselines that we compare against, correlation clustering and average link, which leverage an implementation of NPMI.

**Correlation Clustering (CC).** Widely applied clustering algorithm that penalizes inter-cluster density and intra-cluster sparsity by evaluating pair-wise relationships of tokens and making set-based decision. We use the cautious correlation clustering approximation as described in [BBC04] with $\delta = \frac{1}{44}$.

**Average Link (LINK).** Set clustering algorithm that forms a merge decision based on the best average similarity between all pair-wise relationships of two compared sets while applying agglomerative clustering [CPKT92].

**Greedy Average Link (LINK-G).** Uses the same implementation as LINK but merges sets greedily.

We measure the performance of these techniques through the amount of time it takes to execute them. Furthermore, we measure precision, recall, and f-measure for all token pairs which are common metrics to measure the goodness of a clustering solution [HCML09]. The precision is the number of pairs that are correctly identified as belonging to the same domain while recall represents the percentage of positive ground truth pairs (i.e., record pairs that belong to the same cluster) that have been correctly identified. F-measure then describes the overall quality of a solution with $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

We evaluate these datasets in two different sets of experiments. First, our *threshold experiments* show how performance and quality of our techniques change when we vary the similarity threshold per dataset. In the second set of experiments, the *clustering experiments*, we vary the number of the underlying semantic groups of each input set.

**Threshold Experiments**

For the threshold experiment, we assume two underlying clusters for each input set. The input sets are generated by taking two randomly selected ground truth clusters from each
dataset and combining them. As our algorithms by design are agnostic to whether we are solving the problem of finding single or multiple semantic groups, we thus expect each algorithm to return two clusters. For the Web dataset, we randomly pick 10,000 input sets while combining the ground truth clusters for the Enterprise dataset leads to 4,950 input sets. We make the following observations for this dataset.

**Quality.** Figure 3.32 documents the quality of both datasets when varying the similarity threshold. We generally see that the quality of our algorithms varies depending on the dataset and threshold. For the Enterprise dataset, we observe the best f-measure scores for similarity thresholds 0.6 to 0.8, depending on the algorithm. In contrast, the best
quality results for the *Web* dataset are achieved with similarity threshold 0 to 0.2. This variance is due to the dataset structure as well as the noise contained in the dataset. As the *Enterprise* dataset is smaller, the trade-off between precision and recall becomes overall more favorable with an increase of precision due to the tightening the threshold. For the *Web* dataset, recall drops drastically especially for *Link* whereas GA and GAT remain more stable in their quality degradation.

Overall, we observe that *Link* has slightly better f-measure than both greedy approximative NSC-based approaches. Furthermore, we see that GO has a lower but stable f-measure in both datasets. Recall that the output of GO is independent of the similarity threshold as the only requirement for GO is that the new NSC is higher than the previous NSC scores. Thus, changes to the similarity threshold do not modify the GO clustering progress. Finally, observe that CC has lower performance than *Link*. The reason is that CC makes approximative clustering decisions based on its $\delta$ parameter which results in a faster execution time but a lower average f-measure.

**Performance.** Corresponding to the quality development discussed previously, Figure 3.33 shows the average execution time for the whole dataset per similarity threshold value. Here, the execution time of CC varies depending on the number of executed iterations but is in general lower than the execution time of *Link* due to its greedier decision mechanism. For both GA and GAT, we observe a gradual increase in execution time when the similarity threshold is increased. This increase is steeper for GAT because tightening the threshold decreases the likelihood of exceeding the tight threshold. However, if we compare the execution time of GA and GAT to *Link* for those similarity thresholds that have the highest quality according to Figure 3.32, we see that GAT is on average 8.69 (15.05 for GA) times faster than *Link* in the *Enterprise* dataset and 1.83 (4.1) times faster in the *Web* dataset. Similar to before, the execution time of GO is constant due to its insensitivity of the similarity threshold.

**Greedy vs. Exhaustive Execution Mode.** To differentiate between the impact of the NSC and NPMI scores compared to a greedy or exhaustive execution mode, we took a closer look at the execution time per input set size for those experiments that resulted in the best quality per approach. For example similarity threshold 0.5 guarantees the highest f-measure of 0.95 in the *Enterprise* dataset for *Link* while GAT reaches an f-measure of 0.93 with $\alpha = 0.7$. In Figure 3.34, we show that with an increasing input set size, our tested exhaustive techniques have up to two orders of magnitude higher execution time.
Figure 3.33: Execution time for threshold experiments.

than their corresponding greedy approaches. Specifically, we see that in terms of execution time, Link-G is approximately one order of magnitude faster than Link if the input set size is larger than 80. However, our experiments show that the average quality achieved by Link-G is significantly lower than Link. For Link-G the best quality results are reached with an f-measure of 0.88 which is also far lower than either of the greedy approximative NSC-based approaches achieve. Analogous to the difference between the Link-G and Link, the difference between EA and both GA and GAT is significant (about two orders of magnitude at most). At the same time, it provides only a marginally better f-measure with 0.932.
Figure 3.34: Execution time per input set size based on the best quality results in the threshold experiments.

Summary. Given all of these observations in terms of the quality and performance, we draw the following conclusions for the threshold experiments. First, especially for larger input set sizes which are common when merging data from different data sources, quadratic solutions such as Link are infeasible to compute. Using a) NSC to linearize the similarity computation and b) greedy techniques to improve performance, we show a speed-up of up to two orders of magnitude per input set while only losing 2.6% quality for GAT and 5% quality for GA in this setup. Furthermore, we observe that the performance improvement of applying greedy techniques for NSC-based computation is more significant than for NPMI (see Figure 3.34) while the quality loss is smaller for these approaches. This
suggests that \( NSC \) is a good metric for fast approximations or statistically significant sampling-based techniques.

**Clustering Experiments**

To further explore the trade-off between greedy and exhaustive approaches as well as \( NSC \) and \( NPMI \), we next examine the behavior of Link, Link-G, EA, GA, and GAT in detail for a varying number of underlying clusters. 10,000 sample clusterings are again picked randomly from the ground truth of both datasets as described previously. However, instead of having exactly two underlying clusters, we vary that number from 1 to 5 and examine again the quality and performance trade-off.

**Quality.** In Figure 3.35, we observe a general decrease in f-measure for all approaches with an increase in the number of underlying clusters \( \gamma \). Analogous to previously observed for \( \gamma = 2 \), we see a difference in the quality output between \( NSC \) and \( NPMI \)-based approaches as well as a respective variance in the quality the greedy techniques can achieve. For both datasets, Link provides the best average f-measure while Link-G has the lowest f-measure for the Enterprise dataset and is on average as good as GAT and EA for the Web dataset. We further observe that with a higher \( \gamma \) value, the \( NSC \)-based approaches have a slightly higher quality degradation as Link but they perform better than Link-G. Comparing the greedy \( NSC \) approaches, GAT outperforms GA as expected.

**Performance.** Figure 3.36 shows the execution time for our clustering experiments. Here, we make the following observations. First, all approaches follow a linear increase in execution time when increasing \( \gamma \). Second, GA and GAT are faster than Link by a factor of at least 8 (2.9) and 7 (1.7) for the Enterprise (Web) dataset. Looking at these performance numbers in detail again, we see the same performance characteristics as observed in the threshold experiments. Specifically, we show in Figure 3.37 that to achieve the best quality results with \( \gamma = 5 \), Link computes its high quality results in about two orders of magnitude more time than GA and GAT while Link-G is one order of magnitude slower. This suggests that the performance of the approaches is coupled to \( \gamma \) but the relative difference between these approaches remains approximately the same.

**Summary.** Varying the \( \gamma \) parameter, we show the robustness of our outlier detection techniques and the computational merits of \( NSC \). We further show that albeit decreasing in f-measure for the greedy \( NSC \) approaches, we still observe a f-measure of 0.83 (0.86) for
Figure 3.35: Quality results for clustering experiments.

GAT and 0.76 (0.81) for a high number of underlying clusters, i.e., with $\gamma = 5$, for the Web (Enterprise) dataset. For the same setup, Link achieves a quality of 0.89 (0.95). Given the slightly lower quality but significantly better performance, NSC is a good candidate metric for fast approximations of outliers. In such use cases, quality is not negligible but fast computation time is imperative. For example, we can imagine NSC-based approaches being applied in online spreadsheets for marking entries that do not fit the column while a user is editing the spreadsheet.
Chapter 3: Data Integration Methodology

Figure 3.36: Execution time for clustering experiments.

**Token Corrections**

We now shortly discuss our findings for correcting tokens using NSC and NPMI-based scoring in combination with the JaroWinkler similarity metric as previously outlined in Section 3.2.4. As the JaroWinkler score remains constant, the differences in the generated suggestions and scores is due to the group similarity computation. We furthermore set a threshold of 0.7 and weight the similarity score and group similarity equivalently which means that in order to be considered similar, the NSC or NPMI-based group similarity score needs to be at least 0.4. For these experiments, we extracted the clustering results of CC for $\gamma = 1$, i.e., a use case where CC should have put all records into the same cluster.
Figure 3.37: Execution time per input set size based on the best quality results in the clustering experiments.

For the Enterprise dataset, we observe 30 occurrences were this did not hold, for the Web dataset, we find 730 multi-clusters. Each of these clusterings is expanded through the bipartite graph and suggestions are then evaluated with a scoring function using the NSC or NPMI group similarity metric. We compute NPMI-based group similarity analogous to the Link similarity computation for this set of experiments.

Quality. To understand better why CC did not correctly identify the clustering, we take a look at the clustering alternatives for the outlier tokens. Section 3.2.5 shows a small extract of the corrections that were identified for these two datasets. For example, we
see that token ‘Belgium’ is correctly identified as an outlier with respect to tokens ‘Kiev’ and ‘Brussels’. As corrections, the system then suggests other cities that are syntactically similar to ‘Belgium’. The next two examples shows similar observations, however, for the last clustering we also obtain the original token as suggestion. Token ‘RIMM’ in fact is semantically similar to tokens such as ‘Google’ and ‘Apple’ because it may stand for ‘Research In Motion’ which got acquired by BlackBerry. In this case, CC incorrectly marked ‘RIMM’ as an outlier.

When we contrast the suggestions generated by NSC and NPMI-based techniques, we see that for this setup, the NSC-based scoring mechanism allows us to find more suggestions. The reason is that NSC rewards co-occurrence higher than NPMI which means that for relatively consistent underlying clusters, the support for the suggestion is often bigger than when NPMI is applied.

**Performance.** When comparing the execution time of NSC and NPMI, we observe that on average, the NSC computation is one order of magnitude faster than computing the NPMI score. Specifically, we observe an average execution time of 0.274 seconds for NSC and 1.231 seconds for NPMI on the Web dataset. The maximal execution time for either approach is 10.929 (NSC) respectively 159.383 (NPMI) seconds. An increase in execution time can be observed whenever a large number of suggestions are not pruned from the search space and have to be evaluated. As this mechanism is equivalent between these two approaches, we furthermore observe that when then computing a large number

---

### Table 3.8: Example outliers and suggestions with NSC and NPMI-based scores.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Outlier</th>
<th>Suggestions</th>
<th>NSC</th>
<th>NPMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kiev, Brussels</td>
<td>Belgium</td>
<td>Belgrade</td>
<td>0.8</td>
<td>0.776</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berlin</td>
<td>0.775</td>
<td>0.753</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Beirut</td>
<td>0.724</td>
<td>-</td>
</tr>
<tr>
<td>Queensland, New South Wales</td>
<td>Newcastle</td>
<td>Western Australia</td>
<td>0.716</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Northern Territory</td>
<td>0.707</td>
<td>0.702</td>
</tr>
<tr>
<td>Toronto FC, USA</td>
<td>USA</td>
<td>San Jose Earthquakes</td>
<td>0.717</td>
<td>0.722</td>
</tr>
<tr>
<td>Los Angeles Galaxy</td>
<td></td>
<td>Seattle Sounders</td>
<td>0.7</td>
<td>-</td>
</tr>
<tr>
<td>Yahoo, Google, Apple</td>
<td>RIMM</td>
<td>RIMM</td>
<td>0.817</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM</td>
<td>0.813</td>
<td>-</td>
</tr>
</tbody>
</table>
of group similarities, the linear execution time of $NSC$ significantly improves performance.

**Discussion**

As described in Section 3.2.4, both techniques that we introduce in our work, namely the group similarity metric and its corresponding storage layout, can be leveraged to find adequate outliers and their corrections for use cases such as the one above. Furthermore, we discussed the trade-off between quality and performance of $NSC$ and $NPMI$-based group clustering techniques and showed which integration scenarios favor one over the other. However, we acknowledge that the techniques presented here to address this problem are just of preliminary nature. To find suitable and exact mechanisms for token corrections is out of the scope of our work. Instead, we have focused on introducing a scalable novel metric that can be used for the two use cases highlighted in this part of the dissertation.
Data Integration System Design

In this chapter, we discuss core design ideas for an online event integration system called **StoryPivot** which have been published in [GKRS15] and are described in detail in [GKS16]. It exemplifies a new generation of data integration systems that operate not only on static but can handle continuously changing datasets. We chose event processing as an example domain because it is inherently fast-paced and integrating data in that domain provides valuable information. For example, if we can establish the historical context of events, we are able to show the evolution of events over time and thus find causes for events that occur at a later point in time.

We use **StoryPivot** to show novel design trends for data integration systems which are moving from static to online data processing. That is, data integration systems are now expected to be online systems that propagate updates to the original datasets in near real-time into the integrated solution. As a result, these systems are required to deliver high quality integration in a timely and scalable manner without blocking user requests to the system. For example, engines that deploy entity resolution frameworks are the Facebook Entities Graph [Fac16] and the Google Knowledge Graph [Goo16]. They simultaneously extend their entity base by integrating new crowdsourced or crawled data about entities as more information becomes available while providing access to these entities through their public interfaces. Expanding the first generation data integration systems to online scenarios is a first step in moving towards a next generation data integration system that is designed for online integration. However, we argue that we further need to expand the
The scope of such systems beyond structured, well-defined data such as entities. Specifically, a large percentage of online data is available as unstructured data. Therefore, the next generation of online data integration systems should be able to support unstructured data containing structured concepts such as entities.

The goal of StoryPivot is to integrate real-world events into so-called stories which describe the continuous relationship between events over time. Take as an example the refugee crisis in Europe. Here, news coverage from different data sources connects news articles about the conditions in a Greek refugee camp to interviews of people escaping Syria to information about German immigration politics. The challenge here is to design a system that can process and correlate this kind of events over time and over multiple data sources (e.g., newspapers, blogs etc.). Furthermore, this integration needs to be achieved in a streaming setup because events are continuously reported. There exists a trade-off between the quality of the integrated result and system performance, i.e., how fast data can be integrated, which we examine in our work. We will furthermore show that the developed techniques can benefit a larger set of online data integration applications on structured and unstructured data.

StoryPivot leverages three observations to address the problem of online data integration. First, data within a data source commonly conforms to a source-specific standard and typically has high logical consistency within that source. For example in newspapers, the same real-world event is not covered by multiple news articles unless these articles describe a different angle or additional information becomes available. Thus, there exists an implicit notion of continuity within a data source which can be leveraged to provide high quality integration results over time. Second, if we leverage this source-specific observation, we can parallelize the data integration process. Specifically, StoryPivot first integrates data within a data source which is independent of the data integration process in any other data source. The resulting per-source stories are then integrated across data sources to provide a holistic view. Third, in addition to inter-source parallelization, we leverage intra-source parallelism to reduce the latency of data integration. Our work describes the design decisions that were made when creating StoryPivot and will highlight useful techniques for enabling parallelized, online data integration systems.

To the best of our knowledge, this is the first work that explicitly describes how to construct an online, scalable data integration system from scratch and that discusses which design decisions influence system performance and result quality. With our work, we make the following conceptual contributions.
Online Data Integration. We introduce the notion of continuous data integration systems that are suitable for linking large amounts of data in a scalable manner.

StoryPivot Design. To show the design considerations for a online data integration system, we present a novel event processing and integration system called StoryPivot that provides integrated data for the user through scalable, near-real time data processing mechanisms.

Generalization. We generalize our findings to a wider range of systems that focus on linking data over time.

4.1 Problem Overview

Traditional data integration typically focused on offline data processing, integrating snapshot data, e.g., in ETL processes. In that context, integration commonly consisted of three steps: 1) schema alignment to understand the vocabulary of the integration process, 2) record linkage to determine common entities, and finally 3) data fusion to ensure consistency. StoryPivot can be described as an online linkage system that allows for continuous data integration over time. In contrast to traditional integration techniques, it produces integration results incrementally while using techniques inspired by blocking to improve system performance. Schema alignment in StoryPivot is done as part of the data extraction phase, i.e., we preprocess textual data and annotate it with the same vocabulary across data sources. This allows the system to operate on structured data internally while enabling unstructured data as a potential input source. In the second step, record linkage, our system correlates data within and across data sources. The output of StoryPivot are so-called stories which describe the evolution of real-world events over time. Events are linked but not fused when constructing stories to ensure that source biases are preserved and available for subsequent analyses. However, fusion or summarization of stories is a natural extension of our work.

The goal of StoryPivot is to establish a correlation between real-world events across time and across data sources, forming stories. To accurately represent what happens in the real world, the system first obtains a structured digital representation of different recorded events which we refer to as snippets. It then uses similarity computation and incremental clustering to correlate the snippets and to maintain them and their relationships continuously. These clusters then reflect real-world stories. Computing stories from
events is a novel concept introduced in STORYPIVOT, however, there exist several event processing systems that are designed for event detection and which are closely related to STORYPIVOT and that have been described in detail in Section 2.3 called GDELT [BFS11] and EventRegistry [LFBG14].

To understand the differences between STORYPIVOT and these existing systems, take as an example Table 4.1 which shows snippets that correspond to events extracted between August 12th and 14th 2015 from the ‘Guardian’ (data source $s_1$) and ‘BBC News’ (data source $s_2$). Each snippet is annotated with a timestamp, associated entities, topics, and the title of the article. This information is obtained by taking the underlying unstructured news article and using an open-source NLP framework called NLTK [NLT15] which annotates the article with structured metadata. We refer to each of the categories of the annotated data as a dimension, i.e., there exists an entity dimension, a topic dimension and so on. Then, a snippet $r_{ij}$ denotes the i-th recorded snippet within the j-th data source. As an example, $r_{12}$ is the second snippet recorded in the ‘Guardian’ containing information on the arrival of Syrian refugees on a Greek island called Kos. We now outline the workflow of STORYPIVOT which shows how it integrates snippets and constructs their corresponding stories. This workflow and its components are described in detail throughout the remainder of this dissertation chapter.

Table 4.1: Example of events that are processed by STORYPIVOT.

<table>
<thead>
<tr>
<th>ID</th>
<th>$t_i$</th>
<th>Entities</th>
<th>Topics</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{11}$</td>
<td>Aug 12th</td>
<td>Kos, Refugees</td>
<td>Politics, War</td>
<td>Migrants locked in stadium on Kos</td>
</tr>
<tr>
<td>$r_{12}$</td>
<td>Aug 12th</td>
<td>Kos, Refugees</td>
<td>Politics</td>
<td>Syrian refugees arrive on Kos</td>
</tr>
<tr>
<td>$r_{13}$</td>
<td>Aug 12th</td>
<td>Spain</td>
<td>People, Politics</td>
<td>Bullfighting returns to San Sebastian</td>
</tr>
<tr>
<td>$r_{14}$</td>
<td>Aug 12th</td>
<td>China</td>
<td>Disaster</td>
<td>Tianjin blast sets off earthquake</td>
</tr>
<tr>
<td>$r_{15}$</td>
<td>Aug 13th</td>
<td>China, Tianjin</td>
<td>Disaster</td>
<td>China blasts: hundreds injured</td>
</tr>
<tr>
<td>$r_{16}$</td>
<td>Aug 13th</td>
<td>Greece, Kos</td>
<td>Politics, War</td>
<td>Greece sends cruise ship</td>
</tr>
<tr>
<td>$r_{17}$</td>
<td>Aug 13th</td>
<td>Japan, Tianjin</td>
<td>Disaster</td>
<td>Tianjin explosions visible from space</td>
</tr>
<tr>
<td>$r_{18}$</td>
<td>Aug 13th</td>
<td>Italy</td>
<td>Crime, Politics</td>
<td>Mafia neighbours are bad for business</td>
</tr>
<tr>
<td>$r_{19}$</td>
<td>Aug 14th</td>
<td>Greece</td>
<td>War</td>
<td>Migrants: ‘They said they’d give us papers’</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>Aug 12th</td>
<td>Isis</td>
<td>Politics, War</td>
<td>Islamic State vs Kurds: What’s going on?</td>
</tr>
<tr>
<td>$r_{22}$</td>
<td>Aug 12th</td>
<td>Refugees, Turkey</td>
<td>People, War</td>
<td>Refugees in Turkey: ‘Nothing for us here’</td>
</tr>
<tr>
<td>$r_{23}$</td>
<td>Aug 13th</td>
<td>Refugees, Greece</td>
<td>Politics, War</td>
<td>Chaos amid Greek registration attempt</td>
</tr>
<tr>
<td>$r_{24}$</td>
<td>Aug 13th</td>
<td>Kos, Refugees</td>
<td>People</td>
<td>What happens to migrants arriving on Kos?</td>
</tr>
<tr>
<td>$r_{25}$</td>
<td>Aug 13th</td>
<td>China, Tianjin</td>
<td>Disaster</td>
<td>Tianjin rocked by explosions</td>
</tr>
</tbody>
</table>
4.2: Terminology

**STORYPIVOT Workflow.** Snippets that are added to the system are processed in two steps. First, they are aggregated into clusters within the data source from which they originate. These clusters correspond to real-world stories as seen from the perspective of a single data source. In the second step, source-specific clusters are aligned and integrated across data sources. This allows the system to determine a global view of a story over time and also has the potential to fix data quality issues, for example if a real-world story is split into two clusters in the same source in the first step. An overview of STORYPIVOT is shown in Figure 4.1. Here, the snippets from August 12th have already been processed and have been separated into three different clusters within data source $s_1$ and two different clusters within $s_2$. Furthermore, the system identified one aligned cluster representing a story about the refugee crisis across $s_1$ and $s_2$. Newly arriving snippets throughout August 13th and 14th are managed by a continuously running backend process which applies the two-step integration policy for each snippet. These processes are transparent to the user who should be able to simultaneously access STORYPIVOT for information about stories.

4.2 Terminology

STORYPIVOT is an event processing and integration system that models the evolution of real-world events over time. In this section, we establish the terminology used throughout our work. We establish the mapping of real-world objects and their digital representation as shown in Figure 4.2 and also introduce some of the data structures that we use.
Chapter 4: Data Integration System Design

Definitions

As mentioned previously, the digitalized version of an event is a snippet which is a standardized excerpt of a news article that contains multiple dimensions, i.e., various types of content that describe the event. Going back to the news excerpts recorded in Table 4.1, $r_1^1$ describes the situation of Syrian refugees arriving in Kos as published by the ‘Guardian’ on August 12th 2015. This snippet further contains metadata that can be used for similarity computation such as a title (‘Migrants locked in Stadium’), entities (‘Kos’, ‘Syrian refugees’), or topics (‘Politics’, ‘War’).

Sketches. Let $R = \bigcup R_j$ be the set of snippets that the system observes across all data sources $s_j$. Computing the relationship between all snippets within a data source and across data sources is costly. Therefore, we use the concept of aggregation to efficiently reduce the number of comparison objects in the system within specified time windows. We refer to these aggregation objects as temporal sketches or sketches in short. A sketch $v_i^{j,k}$ represents a set of snippets $r_1^1 \ldots r_l^j \in R$ which occur in the same data source $s_j$ during a time window $t_k$ and have similar semantic content. Each $v_i^{j,k}$ contains the same number of dimensions $d$ as any snippet $r_l^j \in R_j$ though their content may vary depending on the
aggregation mechanism per dimension. **STORYPIVOT** as of now supports full aggregation (all dimensional data from the snippets is completely stored in the sketch) and top-k aggregation. Top-k sketches are summary sketches that do not contain every piece of information but only a selected subset such as the most important $k$ entities or the $k$ words that characterize the text of the snippets best. Our definition of a top-k sketch is loosely based on the notion of bottom-k sketches [CK07]. However, instead of applying a hash function to randomly select representative tokens for each sketch, we apply either cosine similarity with respect to the token’s dimension or frequency counting to determine the top tokens. For example ‘Refugees’ may be a token that appears often in the ‘entity’ dimension of our running example while it is less common in the ‘text’ dimension. As a result, it should have more impact when mentioned in that dimension. The similarity between a sketch and a snippet can then be measured using (weighted) string similarity metrics or cosine similarity across these different dimensions. Details about the aggregation of snippets and its impact on system performance and result quality are further discussed in Section 4.3.

Sketches are the basic computational unit within **STORYPIVOT**. As mentioned above, they are computed for a time window $t_k$ which is a fixed period of time, for example a 24 hour interval which we will use throughout this part of our work in our examples. Within $t_k$, snippets that are semantically similar are integrated into one sketch as shown for the running example in Figure 4.3. Here, $r_1^1$ and $r_2^1$ are similar (they both contain information on the situation of Syrian refugees on Kos) and fall into the same time window, August 12th. They are therefore aggregated into the same sketch.

**Clusters.** If there exist two sketches that are sufficiently similar across time windows in data source $s_j$, we say that they belong to the same cluster $c_l^j$. In our running example, $v_1^{1,t_1}$ and $v_2^{1,t_2}$ as well as $v_3^{1,t_1}$ and $v_1^{1,t_2}$ respectively have similar content in their dimensions which results in them being added to $c_1^1$ ($c_3^j$). Clusters are the digital representation of stories while sketches represent one specific real-world event within a time window $t_k$. Furthermore, we allow sketch hierarchies, i.e., sketches can be combined if they are semantically similar within a time window $t_l$ and their associated time windows $t_{k_i}$ lie within $t_l$. The reason for generating hierarchies is system performance. As mentioned previously, comparing snippets (and sketches) exhaustively is time consuming. Reducing the number of sketches in the comparison space has immediate performance benefits although it may have negative impact on integration quality. We argue that the choice of aggregation level as well as hierarchy is dependent on the environment the system is used in. For example
if communication across data sources is expensive, an intuitive solution is to use sketches that span a bigger time interval to exchange fewer messages. On the other hand, if the system is to use fine-grained sketches within a data source to depict the cluster evolution as accurately as possible, it is not possible to simply adjust the overall sketch size. If hierarchies are applied, the system can be tuned according to the system’s workload which we exemplify for two different datasets in Section 4.5. Finally, an aligned cluster is the result of linking clusters $c_i^{s_j}$ across data sources and represents all knowledge that the system has about a specific story.

**Data Structures**

To internally represent the relationship of sketches as well as the relationship of clusters across data sources, we leverage graph representations. Specifically, we model the intra-source relationship of sketches through a *sketch relationship graph* on top of which a clustering algorithm is applied to find the set of clusters $C_j$ that represent the stories within data source $s_j$. Inter-source relationships of clusters are stored in a *cluster relationship graph*. Applying a (potentially different) clustering algorithm on top of this graph, this graph contains aligned clusters, i.e., it enables us to find stories mentioned in multiple data sources. For the purpose of this work, we use transitive closure as the clustering algorithm of choice for both types of clustering. However, it is part of our future work to
4.3: Integration Quality of StoryPivot

determine whether different, temporally aware, clustering algorithms can further benefit the quality of STORYPIVOT’s integration results.

**Sketch relationship graph.** To identify clusters within a data source, we use a sketch relationship graph $SRG_j = (V_j, E_j)$ where a sketch $v_{jk}^{jl}$ is a node in $V_j$ and an edge label $e(v_{jk}^{jl}, v_{k'l'}^{j'l'}) \in E_j$, with $e(v_{jk}^{jl}, v_{k'l'}^{j'l'}) \in (0, 1)$, is the similarity between two sketches $v_{jk}^{jl}$ and $v_{k'l'}^{j'l'}$ in data source $s_j$. If $e(v_{jk}^{jl}, v_{k'l'}^{j'l'}) = 1$ we say that $v_{jk}^{jl}$ and $v_{k'l'}^{j'l'}$ are identical. If $v_{jk}^{jl}$ and $v_{k'l'}^{j'l'}$ are semantically distinct, $e(v_{jk}^{jl}, v_{k'l'}^{j'l'}) = 0$ holds instead. For creating clusters within $s_j$, the system needs to identify those edges between $v_{jk}^{jl}$ and $v_{k'l'}^{j'l'}$ that indicate that they are similar. We say that there is sufficient evidence for $v_{jk}^{jl}$ and $v_{k'l'}^{j'l'}$ to belong to the same cluster if their similarity exceeds a threshold $\alpha_v$. This methodology is applied in a variety of graph clustering algorithms such as correlation clustering or cut clustering to reduce the clustering search space [BG04]. An example of a sketch relationship graph is the hierarchy level 1 layer of Figure 4.3. Here, an edge exists between $v_{1}^{1,t1}$ and $v_{2}^{1,t2}$ because both of these sketches contain information about entity ‘Kos’ as well as topics ‘Politics’ and ‘War’. Additionally, $v_{2}^{1,t2}$ has an edge to $v_{1}^{1,t4}$ because both of these sketches overlap in the entity dimension and partially overlap in the topics dimension whereas $v_{1}^{1,t1}$ only partially overlaps with $v_{1}^{1,t4}$ in both dimensions. Analogously, there exists an edge between sketches $v_{3}^{1,t1}$ and $v_{1}^{1,t2}$. For the sake of readability, assume in the following that edges take a value of 1 or 0, i.e., there is sufficient evidence that sketches are (not) similar.

**Cluster relationship graph.** A cluster relationship graph is a graph $CRG = (C, H)$ where a node $c_j^l \in C$ represents a source-specific cluster and an edge in the edge set $H$ denotes the similarity between two clusters $c_j^l$ and $c_j^l$, $j \neq l$. In other words, a cluster relationship graph describes the relationship of clusters across data sources. Analogous to the sketch relationship graph, we say that clusters are similar across data sources if they exceed a similarity threshold $\alpha_c$. An example of an aligned cluster is shown in Figure 4.2 on the right-hand side where the two clusters representing the refugee storyline are connected across clusters.

### 4.3 Integration Quality of StoryPivot

We next introduce the two-tier architecture of STORYPIVOT that allows us to efficiently integrate snippets into existing clusters. We call these two tiers cluster identification and cluster alignment.
Cluster Identification

Cluster identification refers to the construction of clusters, i.e., the equivalent of real-world stories, within a data source. To find the cluster that a snippet may belong to, we follow a three-step process. First, we identify the sketches in the system that are similar to the extracted snippet. Second, we merge the snippet with the best match or generate a new sketch if STORYPIVOT did not find a matching sketch. Third, the (new) sketch is compared to other sketches across time windows to identify how similar it is to them. If sufficiently similar sketches are identified, we trigger a reclustering of the sketches.

a) Snippet Matching

As described previously, the content of a snippet can be divided into multiple dimensions. Dimensions, for example a set of entities, are the access points for snippet matching. Specifically, a snippet can only be similar to a sketch if it has overlap with the sketch in one or more dimensions. As an example, recall Table 4.1. After $r_1^1$ is integrated into STORYPIVOT, the system contains exactly one sketch $v_1^{1,t_1}$ with (a subset of) the dimension content of $r_1^1$. When $r_2^1$ is added to the system, it identifies $v_1^{1,t_1}$ as a candidate match because their dimensions are similar (the entities are the same and its topics are a subset of $r_1^1$).

Implementation. To be able to fetch all candidate sketches within the time window corresponding to the incoming snippet, the system deploys a multi-level hash-based data structure. It contains metadata per dimension, time window, and data source that has been observed in the system as shown in Figure 4.4. Indices are not collapsed for one straightforward reason: The distribution of tokens (words, entities, topics etc.) varies depending on the dimension. For example the distribution of words in the titles of these snippets is less constrained than either the topic or entity dimension. As a result, the similarity computation should vary across dimensions which can be achieved by recording dimensions separately. Furthermore, this allows us to parallelize access to the indexing structure: as retrieval usually accesses different parts of the indexing structure, these accesses can be executed in parallel. To reduce the similarity computation space, i.e., to avoid comparing all sketches that match for just one dimension, we use a bloom filter and a threshold for the minimum number of matching dimensions. This filtering mechanism is often helpful as dimensions may have low selectivity. An example is the topic dimension...
in our running example: there are only a limited number of assigned topics. Without the filter, this would cause a significant computational overhead otherwise.

**Example 4.3.1** (Snippet Matching). To retrieve candidate matches, we traverse the index as follows. For each snippet dimension, we extract its tokens, for example \{China, Tianjin\} in dimension Entity for snippet \(r^1_1\). Next, we retrieve all sketches within the same time window that contain the same set of tokens in that dimension. In Figure 4.4, we show that for the Entity dimension, the only match for data source \(s_1\) and token ‘China’ is for timestamp \(t_1\). Analogously, there exists no matching sketch in dimension topic for this snippet. Therefore, the system generates a new sketch, \(v^{1,t_2}_1\), that contains \(r^1_1\) and adjusts the indexing structure accordingly.

b) **Snippet-Sketch Integration**

Whether the system decides to merge a snippet into an existing sketch is dependent on the similarity between the snippet and the sketch, i.e., whether it exceeds the similarity threshold \(\alpha_v\). Let \(D\) be the dimensions for which the system compares the snippet and the sketch. We say that STORYPIVOT merges snippet \(r^j_1\) into sketch \(v^{j,t_k}_l\) if a) the timestamp
of $r^j_i$ is within time window $t_k$ and b) the normalized sum of the similarities across the dimensions is bigger than $\alpha_v$, i.e., if the following requirement holds:

$$\frac{\sum_{d \in D} \text{sim}_d(r^j_i, v^j_{t_k}) \cdot w_d}{\sum_d w_d} \geq \alpha_v$$  \hspace{1cm} (4.1)

Here, we allow different dimensions to be annotated with different weights $w_d$. The reason is that some dimensions have a stronger signal than others for integration purposes. Specifically, dimensions such as entities are effective means of identifying actors in a story where the combination of actors is often a strong signal for the continuity of that story. In comparison, the title of a snippet can be misleading because it is subject to the writer’s creativity. As of now, the weights and the similarity threshold $\alpha_v$ used in STORYPIVOT are manually tested and based on a subset of our data for which we have identified a gold standard manually. Extending this to an adaptive or learned weighting scheme is a part of future research. Ideas in that context encompass per-source modeling of vocabulary and its relevance as well as source-specific weighting schemes.

c) Cluster Construction

In contrast to topic-focused event processing systems such as GDELT [BFS11], the idea behind STORYPIVOT is that stories may evolve over time, i.e., it is possible that not all parts (or in our case: sketches) of the story contain the same dimensional data. Instead, the evolving nature of a story is recorded through the change in the semantic overlap of the dimensions. That is, sketches at time window $t_k$ and $t_l$ can belong to the same cluster even if their metadata is distinct but $t_k \ll t_l$ holds and there exists (at least) a sketch in $t_l$ that is sufficiently connected to the sketches in $t_k$ and $t_l$. As a result, to construct clusters, the system first needs to identify candidate sketches in other time windows that may be related to the current sketch. Afterwards, it computes their similarity and adjusts the clustering solution.

**Implementation.** The identification of candidate sketches that are similar to the new or modified sketch is analogous to the initial snippet matching step. We then compute the pair-wise similarities through a slightly modified version of Equation (4.1), calculating the similarity of two sketches instead of the snippet-sketch similarity. These similarities are then added to the sketch relationship graph. Afterwards, incremental clustering mechanisms as described in Section 3.1.1 are triggered on top of the part of the sketch relationship graph that has been modified. In this phase, one or multiple clusters may have changed
and are thus marked. For each of these marked clusters, the cluster alignment phase is then triggered to readjust the inter-source cluster relationships.

**Example 4.3.2 (Cluster Construction).** In Figure 4.4, the system determined that \( r_{14} \) is assigned to a new sketch \( v_{1}^{t_{1},t_{2}} \). However, there exists another sketch, \( v_{3}^{t_{1},t_{3}} \) that has overlap with \( v_{1}^{t_{1},t_{2}} \) due to its similarity in the entity (‘China’) and topics dimension (‘Disaster’). As a result, we decide in the clustering phase that these two sketches are assigned to the same cluster, \( c_{3}^{1} \), which represents the ‘Tianjin Explosion’ storyline within data source \( s_{1} \).

### Cluster Alignment

Aligning clusters across data sources benefits users in two ways. First, if clusters within some of the data sources are incomplete, they can be enriched when presented to the user as stories. Second, clusters might differ depending on the data source. For example a newspaper that traditional favors democratic politicians will assess a republican debate differently than a republican newspaper. Adding the information of multiple data sources to the output, i.e., presenting data from different data sources to the user, therefore has the potential to improve the knowledge gain of the user.

The computational effort spent on the alignment of clusters across data sources is correlated to the number of data sources that the system uses as input as well as the number of objects that need to be compared per data source. To reduce the time spent on aligning clusters, we introduced the notion of sketch hierarchies in Section 4.2. The idea of sketch hierarchies is to abstract the clusters that have been found but to maintain their temporal character. That is, if cluster summaries are used instead, the evolution over time may not be captured accurately which would thus lead to erroneous alignments. Obviously, any aggregation implies that (part of) the data is omitted. One of the questions that we address in our evaluation (see Section 4.5) is how to find a good balance between aggregation and computational effort.

**Implementation.** Cluster alignment is triggered after the cluster identification phase has been completed. At this point, we know which clusters have been modified. For each of the top-level sketches of each of these clusters, the system then retrieves candidate sketches that match their metadata on the same hierarchy level. For this step, the same index as shown in Figure 4.4 can be used as it already splits the dimensional vocabulary by data source. For all candidate sketches, we then (re-)compute the similarity between the
clusters. Specifically, the similarity is measured as the (weighted) average of all top-level aggregate sketches. If the similarity exceeds a certain threshold, we say that these clusters likely represent the same real-world story. This information is then stored in the cluster relationship graph.

4.4 System Performance of StoryPivot

The core concern when building a large scale online system is to provide high quality and up-to-date data items at any point in time. There is an inherent trade-off between data quality and system performance as more (detailed) data usually leads to better integration results but increases computation time. In the previous sections, we have discussed how we can aggregate data to minimize the number of comparisons between sketches (and summary sketches) that the system has to make. At the same time, our aggregation mechanisms emphasize a source-centric system structure. We further utilize this structure to address the two system challenges that we have described earlier, system latency and scalability. We next discuss how we can a) parallelize parts of our execution pipeline without loss of quality which decreases latency because of faster end-to-end processing times and b) how we can add and remove data sources from the system, thus addressing system scalability.

Pipeline Parallelization

In Section 4.3, we have shown how separating data sources benefits the integration results as the semantic cohesion within a data source is commonly higher than across data sources. Furthermore, we showed how we can improve system performance by aggregating snippets into sketches and finally clusters to decrease system load. Hierarchical compression is not the only measure that StoryPivot takes to improve system latency, i.e., the time it takes to fully integrate a snippet. Leveraging the source-centric data processing model, we discuss two types of parallelization that StoryPivot applies to improve processing time. First, we introduce the notion of intra-source parallelism which is to parallelize the identification of clusters within a data source. Second, inter-source parallelism is the task of parallelizing the computation of aligned clusters across data sources. The computation models for both types of parallelization are shown in Figure 4.5 and are described next.
4.4: System Performance of StoryPivot

Figure 4.5: Intra- and inter-source computation in StoryPivot.

**Implementation of Intra-Source Parallelism.** Recall that in order to evaluate the similarity of a modified sketch with other sketches within a data source, the system compares it to all other candidate sketches that are not in the same time window. Similarity computation is inherently parallelizable as the score between two sketches is dependent on the content of the sketches only. Thus, StoryPivot computes these similarities as follows. For each time window $t_i$ that is different than the time window of the modified sketch $v_{j,t_k}^i$, the system assigns a thread from a pool of available threads that finds those sketches that have an overlap with $v_{j,t_k}^i$. Let $m$ be the total number of time windows currently stored in StoryPivot. We do not need to parallelize the execution for each of these time windows but only for the last $m'$ time windows. Recall that we assume an evolution of events, and thus sketches, over time. To model this evolution, it is not necessary to compare $v_{j,t_k}^i$ with all other sketches $v_{j,t_l}^i$. Instead, for a cluster to evolve, we only need to show that there is an overlap of $v_{j,t_k}^i$’s content with some other sketch $v_{j,t_l}^i$ if $v_{j,t_l}^i$ is in any of the last $\{t_{k-m'}+1, \ldots, t_k\}$ time windows. The number of parallel threads is thus reduced to $m'$ where commonly $m' \ll m$ holds. Afterwards, we collect the results of these similarity computations and synchronize the threads before initiating the next algorithm phase, cluster construction. Synchronization at this point prevents merging the same clusters multiple times. Specifically, it is likely that a new sketch is similar to several sketches that belong to the same cluster if it is similar to one of them. Thus, synchro-
Chapter 4: Data Integration System Design

Figure 4.6: Online integration of data sources in STORYPivot.

nization avoids triggering the same incremental clustering step multiple times. Depending on the cluster assignment, sketches and snippets are shuffled and hierarchical sketches recomputed. Afterwards, cluster alignment is initiated.

Implementation of Inter-Source Parallelism. Similarly to comparing sketches across time windows, their comparison across data sources is inherently parallelizable. That is, the pair-wise similarity of two sketches is not influenced by anything other than their semantic and syntactic relationship. Thus, it is possible to parallelize this kind of similarity computation without decreasing integration quality. In its framework, STORYPivot manages a second threadpool for inter-source parallelization. If a cluster is modified and the alignment phase has been triggered, the system uses one thread per other data source out of that pool to calculate the similarity of the cluster’s sketches with candidate sketches in the respective data source. Once these sketches have been found, their corresponding clusters are derived which is also an operation that can be executed in parallel. Finally, the system recalculates the similarity of the current cluster with each of these clusters and joins the threads to finish the execution.

Scaling StoryPivot

In addition to the parallelization of computational efforts within STORYPivot, its source-centric design enables us to easily add and remove data sources from the system. Take as an example the execution scenario shown in Figure 4.6. Here, snippets $r_1^1$ - $r_4^1$ of data
source $s_1$ have been inserted into the system which at that point contains a set of clusters constructed from these snippets. The integration of data source $s_2$ is now done in parallel to the (streaming) snippet integration of $s_1$: The system batch processes all snippets until it is up-to-date and then proceeds to integrate snippets in a streaming manner. The system load will obviously increase while STORYPIVOT tries to catch up the new data source to the current point in time. However, other data sources that are connected to the system are only minimally impacted by the additional load due to the source-centric design pattern which effectively separates the data sources. As we use separate thread pools for each data source, the computation is shifted to different cores in the machine which minimizes the disruption of the online system.

This way of integrating new data sources is possible if the system load is well balanced and adding another data source to the machine does not limit parallel execution. However, the number of data sources that can be processed with a single-server version of STORYPIVOT is limited. We therefore describe how we plan to scale out STORYPIVOT for an unbounded set of data sources next.

**Distributing STORYPIVOT.** Horizontal partitioning of data sources across multiple servers is possible in STORYPIVOT by design. That is, to achieve load balancing and optimal resource utilization per server, the distributed version of STORYPIVOT, DISTRIBUTED
Chapter 4: Data Integration System Design

**STORYPIVOT**, assigns different data sources to different inter-connected servers. Multiple data sources may also be assigned to the same server as the number of data sources is expected to exceed the number of servers in the system. For example, the *News* dataset used for our experiments in Section 4.5 contains 24,314 data sources thus a balanced distribution of them is imperative. Figure 4.7 visualizes DISTRIBUTED STORYPIVOT. Here, the system consists of multiple servers that are used as storage and computation units. Each of the servers is assigned a (subset of) the data sources that the system manages. DISTRIBUTED STORYPIVOT then handles the following three use cases. First, the integration of a snippet into the system, second, the addition of a data source, and third, user requests. The integration of snippets is handled analogous to the single-server version of DISTRIBUTED STORYPIVOT. The only addition to the distributed system is a central management unit that is able to route snippets to their corresponding data source (here \( r_7 \) to \( s_1 \)). Intra-source computation is solely executed on the server that \( s_1 \) resides on after which inter-source communication is initiated for data sources on the same server as well as across the network. Data source integration leverages the same mechanisms that we described previously. The only difference in a distributed system is the load balancing component that needs to be a part of the system management unit. It decides which server the new data source will be assigned to and will route future requests accordingly. Finally, user requests now do not access a single server but need to be propagated which means that the results of these distributed requests need to be aggregated. We envisage that this aggregation can be either done by the central management unit or on one of the system servers that currently has low load. The key take-away here is that extending STORYPIVOT to a distributed system is straightforward because of its source-centric design characteristics. The description of the distributed system given here is a starting point for such a system extension which will come with a set of new challenges such as load balancing, message passing, and more.

### 4.5 Experimental Evaluation

In this section, we evaluate STORYPIVOT in terms of the trade-off between its end-to-end processing time and result quality. We first describe the setup that we use throughout our experimental evaluation. Afterwards, we examine the quality of our source-centric integration technique. Finally, we discuss the performance of STORYPIVOT and competing event processing techniques.
4.5: Experimental Evaluation

Setup

This section describes the two different datasets that were used for this evaluation as well as implementation details of StoryPivot and the experimental environment.

a) Datasets

For this evaluation, we leverage two real datasets, News and Patents, in multiple variations as follows:

**News.** The News dataset has been collected over 151 days (1st October 2014 - 28th February 2015). It is extracted through the open-source backend of Event Registry [LFBG14] and provides news articles extracted from various sources. Amongst these are well-known newspapers such as ‘The Guardian’, and ‘BBC’, online news portals such as ‘Yahoo News’, and gossip data sources such as ‘PerezHilton.com’. We extract one or multiple snippets per news article by using the NLTK [NLT15] annotation tool. Each of the extracted snippets has five dimensions, a title, associated people, associated organizations, associated locations, and the original text. From the original dataset, we use three variations in this evaluation:

1. News 465k contains 465,321 snippets extracted from 30 different data sources.
2. News 1.2m contains 1,168,233 snippets extracted from 50 different data sources.
3. Obama contains 38,695 snippets that all have the term ‘Obama’ in their people dimension. This dataset is a subset of the News 465k dataset.

The distribution of snippets per day for the first two datasets is shown in Figure 4.8. We observe several patterns in these datasets: for example Saturdays and Sundays usually have a smaller number of observed events, there exist peaks that are caused by important events (right before Christmas (around day 80) significantly more news articles are published), and there are fewer snippets extracted in the mid October - November period than in the beginning of 2015. For the Obama dataset, we observe that the maximum number of processed snippets per day is 1227, the minimum 4, and on average 258 snippets are processed each day.

**Patents.** This dataset has been used to evaluate temporal data integration methods in prior research [LDMS11]. As such, it provides us with the means to evaluate our approach.
in a slightly different setting but where a ground truth is provided such that we can determine the quality of our stories. A story corresponds to the evolution of a person as recorded by her filed name and address for each invention which may change over time. The dataset contains 1925 records with an average of 5.35 records per story.

b) Implementation

We implemented STORYPivot (SP) as a single-server parallelized system as described previously. The implementation is done in Java and all experiments are run on the same server setup, a Linux machine with two Intel Xeon L5520 processors and 24GB RAM.

c) Execution

Our streaming system mimics real-world streaming services. That is, each dataset is evaluated by imitating a streaming behavior such that the execution is equivalent to the original evaluation of the dataset but at a faster speed. Specifically, for the News datasets, we obtain our results by simulating one day of the dataset within one execution hour. For the Obama dataset, we execute one week in one hour as the dataset contains a smaller number of snippets. Finally, we process ten years in a minute in the execution of the Patents dataset as it is comparatively small.
4.5: Experimental Evaluation

Figure 4.9: Comparison of different window sizes in STORYPivot on the Patents dataset.

d) Reference Systems

To compare the execution time of STORYPivot, we implemented a round robin scheme that is a naive alternative to the per-source processing model. In it, every snippet is processed using the same number of threads as used in STORYPivot in a round robin manner. We refer to this processing model in the following as ROUND. Second, we compared our implementation to a sequential processing model (SEQU) using the same story computation algorithms as the parallelized approaches. To compare the result quality of our parallelized approach, we determine the similarity between its result and the event clustering obtained by EventRegistry [LFBG14] and discuss our findings in Section 4.5.

Integration Quality

In this part of our evaluation, we focus on the integration quality of our system. We split our experiments into three parts. First, we examine the Patents dataset to determine the accuracy of our stories when compared to previously established temporal data integration techniques. Second, we compare the output of our system with EventRegistry which applies a domain-based online clustering algorithm [AY06] by manually comparing our stories with their extracted events. Finally, we compare our system to a sequentially executed baseline that applies the same merging policy as STORYPivot but does not differentiate between data sources.
a) Patents Experiments

To the best of our knowledge, the News datasets do not have any associated ground truth which would allow us to evaluate the quality of our system. However, temporal entity resolution is a closely related topic for which several standardized datasets have been introduced over the last few years. The Patents dataset is one of them. It contains data extracted from the European Patent Registry where records are timestamped with the date the patent is filed and they also contain some (limited) information about who filed the patent. Thus, we can use this data to establish the patent filing story of a person. Furthermore, we use blocking as applied in the original temporal record linkage paper [LDMS11]. Specifically, the dataset is annotated such that persons starting with the same letter are processed in the same block. STORYPIVOT is a highly parallelized, sketch-based data integration system. To evaluate its quality, we can vary several parameters such as the degree of parallelization or aggregation. The top f-measure that we observe in our experiments is comparable to the reported f-measure for optimized temporal record linkage techniques as shown in [LDMS11] which indicates that there is no loss of quality because of our parallelized execution framework. We discuss our observations for different experimental setups on this dataset next.

Intra-Source Parallelization. A larger window size, i.e., the span we go back in time to find sketches that may match the current story or sketch, means more computational effort which we verify in our performance evaluation (see Section 4.5 for details). However, the nature of the Patents dataset is that patents are filed on a yearly rather than daily or even weekly basis as we would assume for data such as contained in the News datasets. Thus, we obtain the top f-measure of 0.86 when the window size is five years. The trade-off for different window sizes in terms of precision, recall, and f-measure is shown in Figure 4.9. If the window size chosen is too small, stories cannot be correlated, i.e., the system cannot determine a person’s evolution over time. On the other hand, we might overmatch if the window size is chosen too big because of similarities in the addresses or names of people that were actively filing patents in different periods of time.

Aggregation. We observe that changing the sketch aggregation window, i.e., the number of hours in a sketch, has minimal impact on the f-measure for this dataset. Specifically, we observe a variation of 0.2% in the f-measure score when varying the sketch size from a day, to a week, a month, and so on up to a year.
4.5: Experimental Evaluation

**Blocking.** The blocking mechanism that we use for our experiments is the same as used in previous temporal record linkage work. For comparison, we then evaluated single-source processing for our best-case parallel setup with SEQU. Here, we observe a drastic drop in f-measure to 0.09 due to very low precision scores (0.05). Recall that the applied clustering function is a weighted transitive closure variant. As a result, noise such as different people living at similar addresses at different times may distort the output stories. For example Jean-Pierre Gesson lives in Rue Alexandre Dumas which is the same street Jean-Pierre Goedgebuer lives in. This indicates a very interesting property of a (correctly) applied blocking function: In addition to improving performance, it may also help to improve result precision by providing an implicit noise filter, i.e., a smaller chance to wrongly match records or events. Single-source, sequential execution on the other hand may introduce additional noise as shown in this example and observed in these experiments. We also note that artificially splitting a cluster over multiple data sources does not influence whether or not we can link them. However, they are then connected as aligned stories rather than intra-source stories which is an advantage of STORYPivot compared to traditional blocking algorithms: Our system may recover correlations even if the blocking mechanism made errors in the partitioning.

**(b) Comparison with Sequential Execution**

We compare the identified (aligned) stories of STORYPivot and the sequential processing model in Figure 4.11 for the Obama and News 465k datasets. For the smaller Obama dataset, we observe that the distribution of aligned stories produced by STORYPivot closely resembles the stories identified by sequentially processing the data. This suggests that given a topic-specific dataset (recall that this dataset was generated by filtering dimension *people* for mentions of token ‘Obama’), the parallelization efforts of STORYPivot do not diminish the output clustering. Looking at the more general News 465k dataset, we observe a difference between the found aligned stories in STORYPivot compared to the story distribution of the sequential system. The reason for that is analogous to the observations that we made for the Patents dataset: Noise in the dataset often causes erroneous story merging amplified by transitive closure as clustering method. That is, because clusters are linked with noisy edges, they get merged even though they would otherwise be separated and since there are more options to link a sketch with in SEQU, the level of noise increases. Separating the computation of stories into two phases thus reduces the risk of noise amplification. Because of that, we observe that STORYPivot identifies
244,670 snippets that mostly focus on American politics while the largest story identified with SEQU contains 408,404 snippets where some of them are categorized differently in STORYPIVOT (for example they are then classified as American sports events).

c) Comparison with EventRegistry

For the comparison with the window online clustering algorithm of EventRegistry [AY06], we manually compared the clustering results of the Obama dataset with the clusters proposed by EventRegistry. EventRegistry aims to identify real-world events, i.e., mentions of the same event occurring across multiple data sources. In contrast, STORYPIVOT aims to identify the continuous evolution of events over time and capture them through the concept of stories. We therefore expect that it includes the same clusters as EventRegistry in
Figure 4.12: Throughput and average time for end-to-end snippet processing with varying processing techniques on the News 465k dataset.

the same stories, thus encompassing them. In addition, it should construct clusters across time and align stories across data sources. In our experiments, we observe that StoryPivot identifies 91.52% of the event clusters that have also been automatically computed in EventRegistry. As we do not know EventRegistry’s internal weighting scheme, we use in this experiment a uniformly weighted similarity computation across dimensions people, organizations, and locations. A comparison of the story clusters with EventRegistry clusters (i.e., their notion of an ‘event’), is shown manually through two examples next.

**Greek Financial Crisis.** After a new Greek government came into power in January 2015, Greece started to negotiate their debt payments in early February. These negotiations were a continuous process that developed over several weeks and even lasted multiple months. In the Obama dataset, we observe several extracted snippets in that timeframe that are associated with entities ‘Greece’, or ‘Tsipras’ (the Greek prime minister), discussing these negotiations. For example on February 8th, the Reuters news agency
published an article (captured through snippet $a_1$) that describes how USA is urging the euro zone to compromise with the Greek government. Two days later the Washington Post reports about the new restrictions proposed by the German chancellor ($a_2$). As these two events are close in time, EventRegistry and STORYPIVOT assign them to the same event (resp. aligned cluster). However, the second news article published in the Washington Post was in fact an update of an article that had been published a week before on February 3rd. Comparing its assigned events according to EventRegistry, we observe that it identified different events for $a_1$ and $a_2$. STORYPIVOT on the other hand discovered that the dimensions were a close match and thus assigned the corresponding snippets to the same story. This example shows the fundamental differences of the conceptual elements in either system, the differentiation between events defined for a specific point in time and stories which span longer time intervals.

**US Immigration Laws.** Similar to the previous example, we observe that EventRegistry correctly matches events across data sources within short time windows. For example, the dataset contains two snippets published in USA Today and the New Zealand Herald about the proposed US immigration laws in mid November 2014 ($b_1$ and $b_2$). On November 21st, the Herald then followed up with an interest piece on what these laws would entail ($b_3$) while USA Today published an article about Obama’s point of view of the immigration laws on November 23rd ($b_4$). As $b_3$ and $b_4$ are different in their intent, EventRegistry does not assign them to the same event and fails to draw the connection to $b_1$ for $b_4$ resp. $b_2$ for $b_3$. STORYPIVOT on the other hand not only assigns $b_1$ and $b_4$ ($b_2$ and $b_3$) to the same story but also notes that both stories are related with a similarity across dimensions of 0.357 which is significantly higher than the story alignment threshold with $\alpha_c = 0.1$.

**System Performance**

In this part of our work, we compare the system performance of STORYPIVOT as well as the impact of performance on integration results with competing event processing techniques. First, we show the throughput and average end-to-end integration time of these systems (*End-to-End Experiment*) and discuss performance characteristics for selected data sources (*Data Source Drill-Down*). Afterwards, we vary different parameters to determine the scalability of STORYPIVOT (*Parameter Experiments*).
a) End-to-End Experiment

In this experiment, we varied the event processing model and recorded throughput and end-to-end integration time per snippet. Figure 4.12 shows both performance metrics per processed day of the News 465k dataset. In addition to the streaming execution of STORYPIVOT, we also record the behavior of a batch implementation where two (randomly selected) data sources are added to the system every seven days (i.e., every seven hours in the actual execution) until the system has integrated all data sources.

**Throughput.** All processing models are able to process snippets in a timely manner for the first part of the dataset, i.e., they are able to process incoming snippets within the same time interval they were issued in. For the batch execution of STORYPIVOT, we observe periodic spikes in the throughput of the system every seven days which reflect the catch-up phase of the system. Once the batch version has integrated all data sources, it has the same performance characteristics as the streaming version of STORYPIVOT. When the number of snippets to process becomes large, we see processing difficulties for both ROUND and SEQU. Specifically, comparing STORYPIVOT with SEQU, we observe a higher throughput of up to 2x for the News 465k and up to 4.2x if the same experiment is executed on the News 1.2m dataset. The reason is obviously that the system simply cannot process the sheer number of snippets sequentially and thus fails to integrate them in time. For ROUND, the decrease in system performance is due to parallel accesses to the same portions of the indexing structure. Due to consistency, the same object cannot be used by multiple threads in write operations which becomes a problem if multiple threads want to modify (parts of) the same story.

**Execution Time.** While batch executing STORYPIVOT is the fastest approach initially due to a smaller number of integrated sources, it maintains the same performance as SP after day 105 when no data sources are added to the system any longer. Furthermore, the spikes observed in the throughput measurement are not visible in the execution time which suggests that the integration of new data sources does not interrupt the running streaming integration system. However, we observe a general increase in the processing time per day with a large number of snippets for either SP approach. The reason is again concurrent access to the same objects. As the accessed sketches within the same time interval are constantly modified, threads are blocked when accessing them. This is also the reason for the increased execution time of ROUND. In general, we observe that sequential execution is faster on average because it leverages cache consistency when processing related snippets.
b) Parameter Experiments

We next evaluate how different parameters modify the execution time as well as the distribution of stories in STORYPivot. Specifically, we look at two different parameters, the comparison interval for sketches, i.e., how far back in time we go to find sketches that might belong to the same story, and the sketch size, i.e., the time frame in which information is aggregated.

Varying the Comparison Interval. Figure 4.13 varies the comparison interval \( m' \) from the default \( m' = 15 \) to 7 and 30 days. We observe that with respect to the default execution, decreasing the interval on average improves the end-to-end processing time while increasing the interval worsens performance. Obviously, this is the expected behavior and shown in ?? for the most contested part of the dataset, processing days 100-150. However, note that the difference between \( m' = 15 \) and \( m' = 30 \) (as well as between \( m' = 7 \) and \( m' = 15 \)) is sublinear which suggests that intra-source parallelization is efficient and adds only a small overhead. There exists a difference because we limit the number of concurrent threads in our system to avoid running too many threads in parallel and thus incurring too much thread interference. Next to execution time, we observe a smaller number of overall stories with an increased \( m' \) (see ??). In contrast, decreasing \( m' \) to 7 increases the number of stories, i.e., pieces of stories are kept apart. This is intuitive as a sketch matching across more time windows obviously increases the chance that two sketches are found similar enough to merge the corresponding stories. We further varied \( m' \) to 50 and 100 to observe the impact on the story assignment of events. As expected, we see that the average number of stories per cluster increases, however, we also observe a flattening effect. Specifically, we see that the average number of events per story increases by 9.1% between \( m' = 7 \) and \( m' = 15 \) and only by 5.4% between \( m' = 50 \) and \( m' = 100 \). This suggests that for this dataset, we are able to connect large stories through relatively small time windows.

Varying the Sketch Size. The observations made for the comparison interval are similar to those made for varying the sketch size when it comes to execution time. Here the default sketch time frame \( h \) is twelve hours. In our experiments, we compare it to the execution of \( h = 6 \) and \( h = 24 \). We observe that a larger sketch size decreases the execution time as shown in ?? for days 100 to 150 of the News465k dataset. The reason is that fewer comparison objects are injected into the system as more sketches are merged together. As a result, intra- and inter-source look-ups are more efficient. Furthermore, we observe that the distribution of stories does not change when the sketch size is varied as shown in ??.
c) Scaling StoryPivot

To test the scalability of StoryPivot we compare the results of integrating the News 465k dataset with those obtained from integrating the News 1.2m dataset. In Figure 4.15 we show the execution time for both datasets comparing our system with a sequential execution. We make three observations. First, the average execution time of Sequ is approximately constant over time if the throughput is maximized (2.02 seconds per snippet.

This indicates that for this specific dataset and parameter, there is no negative trade-off between performance and quality.
which allows the system to process 1782.18 snippets per virtual day). As a result of that, executing the \textit{News 1.2m} dataset takes 444 virtual days while it took 165 virtual days to execute the \textit{News 465k} dataset. Second, \textsc{StoryPivot} is not able to process \textit{News 1.2m} within the original 150 virtual days either but requires 206 virtual days less for processing the dataset than \textsc{Sequ}. Around virtual days 120-150, we see an increase in \textsc{StoryPivot}'s processing time because this is the point in time were multiple data sources produce a high volume of snippets. During that timeframe, we observe that the integration of a snippet requires 15.61 seconds. For comparison, the average execution time per snippet in the \textit{News 465k} dataset is 1.91 seconds for this timeframe. However, we observe that during that time, the throughput of \textsc{StoryPivot} is 6986.94 snippets. That is approximately 4 times as much as the sequential algorithm can process within a virtual day (i.e., one execution hour) which is a non-linear improvement over \textsc{Sequ} due to the additional concurrency protocols as well as cache inconsistencies which \textsc{SP} triggers. Third, with nearly 7000 snippets per hour even under high system load, \textsc{SP} can process approximately 150,000 snippets per day. For \textit{News 1.2m}, we observe at most 31,455 snippets in one day which is easily manageable by \textsc{SP} if processed at real-time streaming pace. Sequential processing on the other hand leads to approximately 42,000 snippets processed per day which leaves little space for adding more data sources. Finally, we observe that the execution time increases for highly contested parts of the execution. This behavior is due to the same issue we described above, i.e., data structure consistency as guaranteed by locking mechanisms.
4.6 Discussion

We now discuss how the design decisions made when creating STORYPIVOT can be generalized for a wider range of data integration scenarios. For that purpose, we first summarize the contributions that we make in our work. Afterwards, we will analyze how applicable these design features are for other data integration tasks.

**Source-Centric Design Model.** Our system leverages the observation that processing data within a data source first before integrating it across data sources positively impacts the consistency of stories within the data source while it enables us at the same time to improve end-to-end integration time.

**Scalable Integration Mechanisms.** We showed how integration tasks can be parallelized within and across data sources to improve end-to-end integration time.

**Elastic System Design.** We discussed and showed in our experimental evaluation how SYSTEM-I adjusts to the addition of new data sources online and without interrupting the running system.

These three design features can be extended in a straightforward manner to other online integration systems that are centered around linking records. Take as an example a data cleaning system which uses syntactic cleaning mechanisms. Commonly, data extracted from different source systems has different and source-dependent quality characteristics. Imagine for example two sources capturing information on employees. One of the systems uses a standardization template that shortens employee addresses (i.e., Mission Street becomes Mission St) while the other system uses the full street name. This is a traditional example for an ETL process where integrating the data from both data sources leads to an alignment of the unified data set based on some preset cleaning rules. Implementing the design features that we have discussed before for STORYPIVOT is intuitive for such a system: First of all, it is straightforward that applying a cleaning rule to a dataset can be pushed to each data source without decreasing output quality. Second, this task is parallelizable as each cleaning process can be executed in parallel. Third, alignment across data sources becomes easier as the number of items to compare and consolidate is smaller after in-source standardization.
Data cleaning is not the only integration task that can leverage a source-centric design model but it applies to the more general group of systems that use computationally independent integration mechanisms. An example of interdependent computation is graph processing as seen in Pregel-like systems [MAB+10]. In contrast, traditional data integration systems apply rules, pair-wise similarity computation, or metrics on data items where the item’s computation is independent of the computational efforts the same system makes on a different data item. Using mechanisms such as blocking that partition the search space (analogous to the time windows within STORYPIVOT), we can leverage parallelization within a data source. Additionally, if there exist multiple data sources, inter-source parallelism can be applied. For this type of integration systems, the design characteristics that we have exemplified through STORYPIVOT can be implemented with minimal overhead.
Conclusion

This dissertation discusses and advances data integration methodology and design aspects for data integration systems under the assumption of dynamically changing data sources. We compared our work with traditional data integration systems which solved tasks such as entity resolution and data cleaning for static datasets and show how we can a) modify existing data integration techniques to accommodate changing datasets, b) use novel data sources such as the crowd effectively, and c) enable real-time interaction with data integration tasks. All of these techniques were discussed in detail in the first part of this work, data integration methodology. The second part of our work then focused on the system design of online data integration systems which we showed by implementing our own event integration system called STORYPivot. We now shortly summarize the contributions in both of these areas and point out future research opportunities.

5.1 Data Integration Methodology

Our contributions in this area can be split into three parts corresponding to the three key take-aways pointed out above. The first part of our work exemplified through the task of entity resolution how graph-based clustering techniques can be adapted to incremental changes in the dataset, i.e., changes that do not affect a majority but a minority of the nodes in the graph. For that purpose, we examined existing clustering techniques such as
correlation clustering and DB-index clustering and showed which properties are required to maintain high quality results while improving integration performance for (partial) dataset updates. We then proposed two optimal algorithms that can be applied to clustering techniques fulfilling these properties and furthermore introduced a greedy approximation of one of these algorithms that has good quality and performance characteristics as empirically shown in our experimental evaluation.

Aftwarwards, we extended our work on entity resolution to encompass entity resolution methods for crowdsourced data. Crowdsourcing is a data source that is often unpredictable because the quality of a crowd worker is dependent on a variety of parameters such as the worker’s motivation, experience, or attention to the task. We thus developed and implemented a graph-based strategy to determine for pair-wise entity resolution whether or not two items point to the same real-world entity. This technique is then applied for the iterative and incremental construction of an ER solution where we use the techniques developed previously in our work in a new environment. Our work furthermore points out several strategies that address the next-crowdsourcing problem which is the task of choosing the best possible question to ask the crowd. These strategies can be applied for any kind of binary crowdsourcing comparisons that involve the clustering of items.

Next to entity resolution, our work on data integration methodology additionally examines another traditional data integration task, data cleaning. Similar to entity resolution, cleaning (potentially) dirty data is often done as an offline process. Given an online integration environment, we thus ask the question of how we can facilitate fast but correct identification of which items need to be cleansed given an input set of items. In practice, we envision that these items are for example values that a user enters in a column in an online spreadsheet. Intuitively, there exists some notion of consistency between such columnar values which is what we mathematically capture through the notion of group similarity. We show that group similarity computation is an efficient alternative to previously applied pair-wise techniques and achieves good quality estimates with a performance improvement of up to two orders of magnitude in our experimental setup.

**Future Work.** There are some methodological challenges that directly follow from the techniques developed in this dissertation. For example, we outline in our work on data cleaning how we can generate corrections for tokens on the fly. Specifically, we propose to leverage our bipartite relationship graph to prune the search space and then leverage a combination of group similarity computation and semantic similarity to determine candi-
5.2: Data Integration System Design

data corrections. This requires a) an accurate bipartite relationship graph and b) accurate similarity computation methods. However, both of these problems are not yet solved as our bipartite relationship graph is parsed from web data and can thus contain potentially erroneous data. At the same time, some similarity metrics may be more useful for some cleaning scenarios than others. Improving and then choosing the combination of these two techniques is thus an open research question.

Similarity metrics are not only applied in data cleaning but are a crucial part of the entity resolution process as well. In our work on crowdsourced entity resolution, we have shown how these kind of automatic similarity functions can be combined with input from human data sources to improve integration quality. The space of combined automatic and crowdsourced data integration techniques is still an open area of research where techniques commonly address one integration problem at a time. As a result, it would be interesting to determine and generalize which tasks benefit from this interactive model, which can still be improved upon, and which we could consolidate into a single interactive data integration system. In addition, there are several open research questions in the context of crowdsourcing that this dissertation only shortly touches upon because they are out of scope of our contributions. An example for these are determining the quality of automatic and human data sources accurately and using these findings in the data integration processes.

5.2 Data Integration System Design

As explained earlier, traditional data integration systems manage and integrate static datasets. Adapting the algorithms and mechanisms applied in these systems to dynamic data sources is a first step in transitioning from static to online data integration but it also implies that for each algorithm, the system developer needs to revise and redesign them. As an alternative, our work discusses key system design elements that are especially helpful for the management and integration of large-scale data linkage scenarios. We develop these elements by designing our own event integration system which takes as input data from various data sources and integrates the data in a two-step process into stories which represent the correlation of events over time and potentially across data sources. Our work touches upon the trade-off between quality and performance of fast-paced data integration systems as they may abstract from data streams and then (approximately) integrate data using parallel execution mechanisms. It is important to note that we show
how abstraction, for example aggregation, and parallelization can be implemented in a way that minimizes the interaction across parallel threads and at the same time maintains high quality integration results. Specifically, we use the observations made previously for incremental data integration and use them to maintain up-to-date integration results in near real-time. The design ideas behind StoryPivot are not limited to event integration only as we outline in Section 4.6. Instead, we envision that the applied methodology can be applied to similar multi-source data linking problems.

**Future Work.** The StoryPivot project not only touches on data integration topics but it is by design a system for online event integration which users should be allowed to interact with. More specifically, StoryPivot uses a customized inverted index to map the content of a snippet to the current set of sketches. One of the positive side effects of this design decision is that user queries can use the same access patterns when requesting stories from the system. Given that an aligned cluster can span multiple data sources and large time windows, an obvious question to address in this context is how clusters are visualized once the relevant sketches have been retrieved. We have shown in the experimental evaluation that clusters may contain hundreds of thousands of snippets. Thus, summarization as well as prioritization are important research aspects if StoryPivot would be commercialized.

Obviously, StoryPivot also enables other research in the area of data integration. For example, the clustering techniques for story identification that are currently employed in the system are relatively simple. However, temporal record linkage is a topic that has been studied in recent years in the context of entity resolution [CDN14, LDMS11]. Here, the idea is that data may change over time which is the same notion that we associate with stories. One area of research that follows up on our work is how to do the (incremental) clustering process that we propose for connecting sketches more accurately. So far, most of the similarity-based computations are based on well-established string metrics. However, as we have pointed out earlier, dimensions may vary in their content (per snippet). Thus, an interesting direction for future research would be to learn how to correlate snippets across dimensions when trying to maximize result quality for various temporal, incremental clustering techniques.
List of Tables

3.1 Example for merge operation. ............................................. 55
3.2 Statistics of real-world datasets according to CAUTIOUS. .............. 60
3.3 Comparison of various algorithms on Biz. Highest performance is highlighted in bold. Penalty values are averaged. Improvement is calculated for NAIVE w.r.t. BATCH and for the other methods w.r.t. NAIVE. .......... 64
3.4 Comparison of CC, IT, and GREEDY for varying update sizes and the Biz dataset using the CONT incremental execution strategy. .................. 66
3.5 Details on Biz (Correlation Clustering, CONT). ............................. 67
3.6 Algorithm comparison for Cora. F-Measure is averaged. ................. 67
3.7 Datasets Overview .......................................................... 101
3.8 Example outliers and suggestions with NSC and NPMI-based scores. .. 146
4.1 Example of events that are processed by STORYPIVOT. .................. 152
## List of Figures

1.1 Static processing of business listings. .................................................. 2
1.2 Dynamic processing of business listings. .............................................. 3
1.3 Dynamic web table modification example. ........................................... 4
1.4 Overview of contributions. ................................................................. 10

2.1 Overview of data integration process. .................................................. 15
2.2 Overview of different ER strategies. .................................................... 31

3.1 Naive clustering based on Figure 1.1 (a) and Figure 1.2 (a). ................. 37
3.2 Desirable entity resolution properties. .................................................... 42
3.3 Relationships between the properties. ................................................... 44
3.4 Violation of properties through DB-index. ............................................ 47
3.5 Iterative algorithm for Example 3.1.4 - Part 1. .................................... 52
3.6 Iterative algorithm for Example 3.1.4 - Part 2. .................................... 52
3.7 Execution time comparison for CONT on Biz dataset. ........................ 65
3.8 Penalty comparison for CONT on Biz dataset. ..................................... 65
3.9 Experimental results on Cora dataset. .................................................. 68
3.10 Experimental results on Febrl dataset. ............................................... 70
3.11 Execution time for varying synthetic datasets. .................................... 72
3.12 Animal classification problem. ........................................................... 75
List of Figures

3.13 MINMAX computation examples. 83
3.14 Incremental update propagation. 88
3.15 MINMAX computation time (synthetic dataset, perfect crowd). 92
3.16 Queuing strategies for Landmarks dataset (synthetic crowd). 97
3.17 Parallelization problem. 98
3.18 Data interpretation for Landmarks dataset (synth. crowd, part 1). 103
3.19 Data interpretation for Landmarks dataset (synth. crowd, part 2). 104
3.20 Strategy evaluation for Landmarks dataset (real-world crowd). 106
3.21 Execution time for Landmarks dataset (real-world crowd). 107
3.22 Parallel execution for Landmarks dataset (real-world crowd). 108
3.23 Data interpretation for Publications dataset (real-world crowd). 109
3.24 Strategy evaluation for Publications dataset (real-world crowd). 110
3.25 Parallel execution for Publications dataset (0.3 < s < 1, real-world crowd). 111
3.26 Overview of the different framework components and their interactions. 121
3.27 Exploration of TS graph. 123
3.28 Scaling effects for OI and COI. 124
3.29 Group similarity computation using pair-wise and group similarity metrics. 125
3.30 NSC score development for varying group with p(r_i)=0.05. 128
3.31 Example for token correction. 134
3.32 Quality results for threshold experiments. 138
3.33 Execution time for threshold experiments. 140
3.34 Execution time per input set size based on the best quality results in the threshold experiments. 141
3.35 Quality results for clustering experiments. 143
3.36 Execution time for clustering experiments. 144
3.37 Execution time per input set size based on the best quality results in the clustering experiments. 145

4.1 Workflow of STORYPivot. 153
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Mapping of real-world to virtual-world terminology in STORYPIVOT.</td>
<td>154</td>
</tr>
<tr>
<td>4.3</td>
<td>Relationship between snippets, temporal sketches, and clusters for snippets.</td>
<td>156</td>
</tr>
<tr>
<td>4.4</td>
<td>Partial index for dimensions ‘entity’ and ‘topic’ for snippets ( r_1^1 - r_1^2 ) and ( r_2^1 - r_2^2 ).</td>
<td>159</td>
</tr>
<tr>
<td>4.5</td>
<td>Intra- and inter-source computation in STORYPIVOT.</td>
<td>163</td>
</tr>
<tr>
<td>4.6</td>
<td>Online integration of data sources in STORYPIVOT.</td>
<td>164</td>
</tr>
<tr>
<td>4.7</td>
<td>STORYPIVOT as a distributed system.</td>
<td>165</td>
</tr>
<tr>
<td>4.8</td>
<td>Distribution of snippets in the News 465k and News 1.2m datasets (stacked).</td>
<td>168</td>
</tr>
<tr>
<td>4.9</td>
<td>Comparison of different window sizes in STORYPIVOT on the Patents dataset.</td>
<td>169</td>
</tr>
<tr>
<td>4.10</td>
<td>Aligned Stories (Obama)</td>
<td>172</td>
</tr>
<tr>
<td>4.11</td>
<td>Distribution of (aligned) stories for STORYPIVOT and sequential processing</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>on the Obama and News 465k datasets.</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>Throughput and average time for end-to-end snippet processing with varying</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>processing techniques on the News 465k dataset.</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>Comparison of different window sizes in STORYPIVOT on the News 465k dataset.</td>
<td>177</td>
</tr>
<tr>
<td>4.14</td>
<td>Comparison of different sketch sizes in STORYPIVOT on the News 465k dataset.</td>
<td>177</td>
</tr>
<tr>
<td>4.15</td>
<td>Execution time of STORYPIVOT on both News datasets.</td>
<td>178</td>
</tr>
</tbody>
</table>


Bibliography


Bibliography


Curriculum Vitae

Anja Gruenheid

Affiliation During Doctoral Studies

09/2012 – 09/2016  Research Assistant
Department of Computer Science, Systems Group
Swiss Federal Institute of Technology Zurich, Switzerland.

Education

04/2010 – 08/2012  Master of Science, Computer Science
Technische Universität München, Germany.

08/2010 – 05/2011  Master of Science, Computer Science
Georgia Institute of Technology, USA.

09/2006 – 09/2009  Bachelor of Science, Computer Science
Technische Universität München, Germany.

Work Experience

09/2015 – 12/2015  Development Intern
Google, USA.

06/2015 – 08/2015  Research Intern
Microsoft Research, USA.
Curriculum Vitae

02/2012 – 08/2012  **Research Intern**  
*AT&T Labs Research, USA.*

08/2009 – 12/2011 **Development Intern**  
*IBM Deutschland Research & Development GmbH, Germany.*

*Preparation of a Start-Up Company, Germany.*

**Teaching**

2012 – 2016 **Co-Leader Forum for Women in Computer Science**  
*ETH Zurich, Switzerland.*

Spring 2016 **Data Modelling and Databases**  
*ETH Zurich, Switzerland.*

Fall 2013 **Introduction to Programming**  
*ETH Zurich, Switzerland.*

Fall 2012 **Information Retrieval**  
*ETH Zurich, Switzerland.*

**Selected Publications**

2015  
A. Loro, A. Gruenheid, D.Kossmann, D. Profeta, P. Beaudequin  
*Indexing and Selecting Hierarchical Business Logic*  
PVLDB 8(12): 1656-1667, 2015

A. Gruenheid, T. Rekatsinas, D. Kossmann, D. Srivastava  
*StoryPivot: Comparing and Contrasting Story Evolution*  
SIGMOD Conference: 1415-1420, 2015

2014  
A. Gruenheid, X. L. Dong, D. Srivastava  
*Incremental Record Linkage*  
PVLDB 7(9): 697-708, 2014
2013  
A. Gruenheid, D. Kossmann  
*Cost and Quality Trade-Offs in Crowdsourcing*  
DBCrowd Workshop, 2013

2011  
A. Gruenheid, E. Omiecinski, M. Leo  
*Query Optimization Using Column Statistics in Hive*  
International Database Engineering and Application Symposium, 2011