8-2018

Online Data Cleaning

Elkindi Rezig
Purdue University

ONLINE DATA CLEANING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Elkindi Rezig

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Walid G. Aref

Department of Computer Science

Sunil Prabhakar

Department of Computer Science

Christopher W. Clifton

Department of Computer Science

Sonia Fahmy

Department of Computer Science

Mourad Ouzzani

Qatar Computing Research Institute

Ahmed K. Elmagarmid

Qatar Computing Research Institute

Approved by:

Voicu Popescu / William J. Gorman

Head of the Computer Science Graduate Program

To my parents and my deceased grandfather Salah

ACKNOWLEDGMENTS

First and foremost, all thanks go to Allah for giving me the strength and patience to complete my Ph.D.

Throughout my Ph.D. years, I had the chance to work closely with Prof. Mourad Ouzzani and Prof. Walid Aref who jointly served as my Ph.D. advisors. Both of them took the time and care to guide every step I took in my research endeavours. Through their continuous guidance and support, I learned and grew up to be the person I am today. Even though Prof. Ouzzani works thousands of miles away from Purdue, he was always available to talk to me over Skype anytime I needed to reach out to him. His astute attention to detail was crucial to take my ideas across the finish line. I feel immense gratitude for his tireless guidance and mentorship from my first day at Purdue until my graduation. My other advisor, Prof. Aref, has an uncanny ability to express complex, technically-deep material using an incredibly clear language. His clarity of thought has always inspired me. Every meeting I had with him was invigorating to me at the professional and personal levels. No words will do justice to the gratitude I feel towards Prof. Mourad Ouzzani and Prof. Walid Aref for all the assistance they offered me throughout the years.

I would like to thank Prof. Ahmed Elmagarmid for always finding time in his busy schedule to give me feedback on my work. His comments expanded my thinking and stimulated me to produce ideas that are useful inside and outside the confines of academia. I feel deeply honoured to have worked with a scholar of his caliber.

I was very fortunate to closely collaborate with Prof. Eduard Dragut in my early Ph.D. years. Back then, Eduard was a postdoc at Purdue. I have learned a lot from him and he was always available to answer any questions I had. My collaboration with him was key in my Ph.D. and I will be forever grateful to him.

I am very grateful to Prof. Gustavo Rodriguez-Rivera who trained me to become an effective teacher of several computer science classes. Gustavo is an excellent instructor in

the department of Computer Science at Purdue. He was able to spread his teaching passion to me and I will always be grateful for all the time he spent mentoring me throughout my teaching career.

I thank my family in Algeria for always supporting me. My mother Turkia Rouag, my father Abdelkader, my sister Asma and my brother El Razi. My family members have always been my cheerleaders throughout my life. Through good and bad times, they never ceased to look after me. Their support has always fuelled any effort I undertook.

My graduate life would have been much more stressful, had it not been for the supportive social circles I had the chance to be a part of. Particularly, I would like to thank the Islamic Society of Greater Lafayette (ISGL) for creating a social hub to the Muslim community in Purdue. Within the confines of the ISGL building, I was able to forge relationships that blossomed into true, lasting friendships. I thank Salah, Haroon, Malick, Harsh and Abdelrahman for making my life at Purdue much more joyful.

Last but not least, I would like to offer my sincere gratitude to my Ph.D. examining committee members for their diligent comments and commitment to assess my work. Their continuous commitment to help students succeed is what makes Purdue a higher education leader.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Rezig, Elkindi PhD, Purdue University, August 2018. Online Data Cleaning . Major Professors: Walid Aref and Mourad Ouzzani.

Data-centric applications have never been more ubiquitous in our lives, e.g., search engines, route navigation and social media. This has brought along a new age where digital data is at the core of many decisions we make as individuals, e.g., looking for the most scenic route to plan a road trip, or as professionals, e.g., analysing customers' transactions to predict the best time to restock different products. However, the surge in data generation has also led to creating massive amounts of dirty data, i.e., inaccurate or redundant data. Using dirty data to inform business decisions comes with dire consequences, for instance, an IBM report estimates that dirty data costs the U.S. $3.1 trillion a year.

Dirty data is the product of many factors which include data entry errors and integration of several data sources. Data integration of multiple sources is especially prone to producing dirty data. For instance, while individual sources may not have redundant data, they often carry redundant data across each other. Furthermore, different data sources may obey different business rules (sometimes not even known) which makes it challenging to reconcile the integrated data. Even if the data is clean at the time of the integration, data updates would compromise its quality over time.

There is a wide spectrum of errors that can be found in the data, e,g, duplicate records, missing values, obsolete data, etc. To address these problems, several data cleaning efforts have been proposed, e.g., record linkage to identify duplicate records, data fusion to fuse duplicate data items into a single representation and enforcing integrity constraints on the data. However, most existing efforts make two key assumptions: (1) Data cleaning is done in one shot; and (2) The data is available in its entirety. Those two assumptions do not hold in our age where data is highly volatile and integrated from several sources. This calls for a

paradigm shift in approaching data cleaning: it has to be made iterative where data comes in chunks and not all at once. Consequently, cleaning the data should not be repeated from scratch whenever the data changes, but instead, should be done only for data items affected by the updates. Moreover, the repair should be computed efficiently to support applications where cleaning is performed online (e.g. query time data cleaning). In this dissertation, we present several proposals to realize this paradigm for two major types of data errors: duplicates and integrity constraint violations.

We first present a framework that supports online record linkage and fusion over Web databases. Our system processes queries posted to Web databases. Query results are deduplicated, fused and then stored in a cache for future reference. The cache is updated iteratively with new query results. This effort makes it possible to perform record linkage and fusion efficiently, but also effectively, i.e., the cache contains data items seen in previous queries which are jointly cleaned with incoming query results.

To address integrity constraints violations, we propose a novel way to approach Functional Dependency repairs, develop a new class of repairs and then demonstrate it is superior to existing efforts, in runtime and accuracy. We then show how our framework can be easily tuned to work iteratively to support online applications. We implement a proof-of-concept query answering system to demonstrate the iterative capability of our system.

1 INTRODUCTION

Businesses are motivated more than ever to turn their data into insights. For instance, a company devising an advertising strategy for a product should first identify the customer demographics it should target, i.e. customers who are likely to purchase the product. Recent advances in data science have incentivized the race to collect massive amounts of data for analysis. However, data analysis is largely compromised by dirty data. As a result, businesses have to choose between two harsh choices: (1) analyze inaccurate data and end up with potentially faulty decisions; and (2) invest massive amounts of resources to tackle the problem of data quality. An Expedian report [1] published in 2015, estimated that 32% of U.S. companies feel their data is inaccurate. As a result, dirty data comes with a hefty financial loss for businesses, for instance, an IBM report estimates that U.S. companies lose $3.1 trillion a year [2].

Accurate data is condusive to successful business decisions. However, data curation is often seen as a by-product of another task, e.g., data analytics. In many cases, humans whose primary task is not to clean the data often find themselves cleaning the data as a preliminary step to perform their job. For instance, it has been reported that data scientists in Merk, a large pharmaceutical company, spend 98% of their time preparing and curating the data [3]. This only leaves them with 2% of their time to perform analytical tasks. Software engineers are also often burdened with the task of cleaning the data before writing programs that process it [4,5]. Therefore, it is crucial to have tools that identify data errors, and possibly correct them.

Data errors come in different types and are often hard to even detect. Figure 1.1 illustrates an example table T_1 containing several types of errors that include outliers (the salary in $t_3[Salary]$ differs largely from the other salaries), duplicates (t_4 and t_5 are deemed duplicates), integrity constraint violations (t_1 and t_2 have cells that violate integrity constraints) and formatting errors ($t_4[City]$ should follow the same format as $t_5[City]$). Detecting the

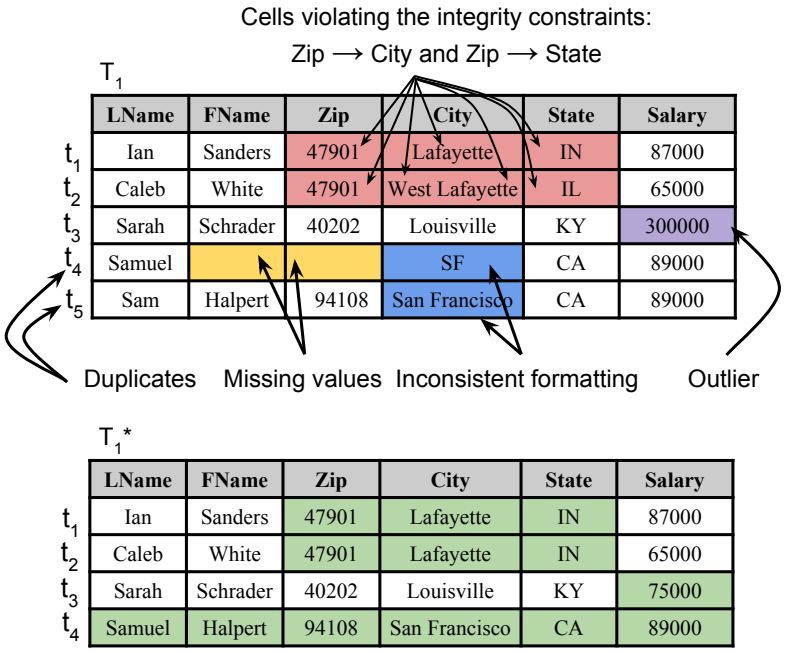


Figure 1.1.: Example of data errors and possible repairs for them

many types of data errors is by itself a notoriously difficult problem that has attracted a lot of attention [6–10].

1.1 Data Cleaning at a Glance

Data cleaning is the process of detecting data errors, and then repairing them. This process is conducted by tools, humans or both. Because errors can come in a variety of flavours, data cleaning is a multi-faceted process often characterized by: (1) the type of data errors we are trying to detect, e.g., duplicates; (2) how errors are fixed, e.g., how to fuse two duplicate records into one record; and (3) whether humans are involved in the data cleaning loop, e.g., errors in salary data can be fixed by humans only.

Error detection is the first step of the cleaning process. There are different ways to detect errors depending on their type. For example, to detect rule violations, one could use integrity constraints (e.g., functional dependencies) to express business rules the data has to obey (e.g., two records that share the same zip code must share the same city name). To

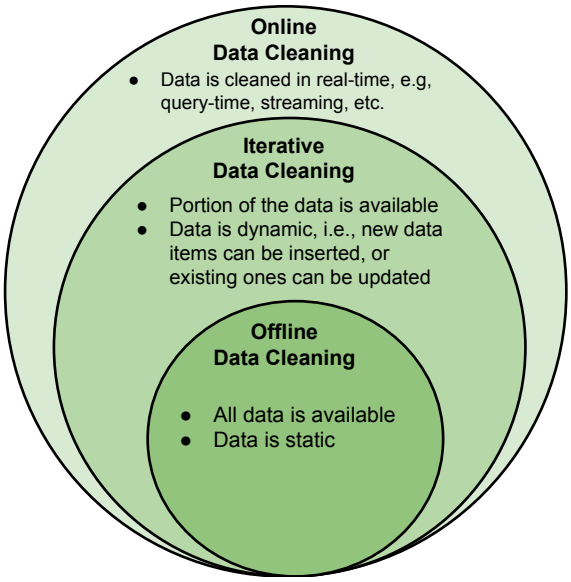


Figure 1.2.: Classification of data cleaning settings

detect duplicate records, similarity measures (e.g., string similarity) are typically used to decide if two records are potentially duplicates [11].

Given the detected errors, the data repairing step aims at fixing those errors. This phase is typically conducted by "helper" data cleaning tools whose output is then checked by a human. There is a plethora of efforts that aim at providing data repairs that are accurate and require minimal human intervention [7, 12].

In Figure 1.1, a possible repair for instance T_1 (top table) is illustrated in table T_1^* (bottom table). The cells that were involved in the repair are highlighted. The repair could be computed by a variety of actors, including humans and tools. Repairing different types of errors often involves several tools and human actors [7, 8]. However, since the human resource is expensive, most data cleaning techniques strive to judiciously involve humans to detect and repair errors [9]. It is clear that scaling data cleaning to larger datasets [13] requires tools that are able to do the bulk of the cleaning effort, e.g., it is infeasible to ask humans to clean millions of records.

1.2 Online Data Cleaning

There are several classifications of data cleaning techniques in the literature [14, 15], e.g., classification according to who cleans the data, the type of errors, etc. We present a novel classification of data cleaning techniques according to three key parameters:

1. **Data availability:** Many data cleaning techniques assume that the data is available at its entirely. These techniques fall short when only a portion of the data is available to the algorithm.

2. **Data volatility:** In many scenarios, data is not static, i.e., it can include new chunks, or existing items are updated over time. Few efforts deal with this setting.

3. **Cleaning time requirements:** Some emerging applications, e.g., search engines that integrate results from several sources, require not only data cleaning at query-time, but they also require it to be conducted in real-time (or online). As a result, this setting does not tolerate long cleaning times.

Figure 1.2 illustrates a classification of data cleaning approaches according to the parameters outlined above. We distinguish the following classes of data cleaning techniques:

1. **Offline data cleaning:** Techniques that fall in this class assume the data is available in its entirety and does not change. Most existing data cleaning techniques fall into this class [16–18].

2. **Iterative data cleaning:** In this setting, the data cleaning algorithm has to be able to efficiently support adding new data or updating the existing data. This class of techniques does not assume all the data is available and supports dynamic data [19–22].

3. **Online data cleaning:** In this setting, the techniques not only have to be able to clean the data offline and iteratively, but should do so efficiently. In this context, portions of the data are requested on-demand, e.g., at query time [19, 20], streaming [23].

1.3 Challenges in Online Data Cleaning

Moving data cleaning to the online setting presents several challenges that pertain to the quality of the produced repaired data (accuracy challenge) and the time it takes to fix the errors (efficiency challenge). Even in the offline setting, both of these challenges remain largely unsolved. Therefore, the online setting significantly exacerbates the difficulty of data cleaning.

1.3.1 Accuracy Challenge

It is clear that cleaning partial data inherently leads to erroneous cleaning decisions. For instance, if we adopt a majority-voting strategy to resolve a violation (e.g., choosing the the most frequent city name to resolve a violation to the FD $Zip \rightarrow City$), then, a majority value in one instance of the data may no longer hold after the data is updated. Therefore, cleaning the data at time t provides little help to clean it again at tine $t+1$ (after it has been updated).

1.3.2 Efficiency Challenge

In many data-intensive applications, it is important to have part of the data cleaned on-demand. For instance, data scientists using the company data warehouse would need to post queries that pertain to small fractions of the data in their analytical task. Therefore, in the online setting, the data is not cleaned in its entirety, only the parts that are returned to the user are.

1.4 Contributions and Outline

In this dissertation, we present contributions that address challenges in all the data cleaning classes illustrated in Figure 1.2 and for several data errors. Specifically, our contributions are outlined as follows:

- **Query-time deduplication and fusion. [19,20]** This work addresses challenges that pertain to Iterative and Online Data Cleaning. We propose an end-to-end framework that supports query-time record linkage and fusion. As an example application that needs Online Data Cleaning, we implemented a virtual integration system (VIS) that queries multiple Web databases using their respective Web forms. In this application, the VIS can only access the data through queries (no access to entirety of databases), data comes in chunks (query results) and the query has to be answered fast. The data is deduplicated using off-the-shelf offline deduplication and fusion techniques.

- **Pattern-driven functional dependency repairing. [24]** This work addresses challenges in Offline Data Cleaning. We proposed a new way to approach the problem of FD repairing. The current FD repairing algorithms tightly couple detection and repairing. That is, the set of violating cells are identified and then repaired. However, when not all the data is available, this coupling falls short, i.e., the repaired cells in one data instance may violate new cells in another data instance. Therefore, saving the repaired instances does not offer any advantage in quality, i.e., if a wrongly repaired instance at time t is jointly repaired with new incoming data at time $t+1$, then, the new data items will also be wrongly repaired. Consequently, we could not simply use off-the-shelf FD repairing algorithms and adapt them to work in the iterative and online settings. We had to rethink the whole process of repairing FD violations from scratch with the iterative and the online settings in mind. Our proposal decouples data representation from its repairs. That is, we do not store repaired instances of the data and iteratively repair them with new data items, instead, we propose a novel way to identify and model values combinations, referred to as patterns, by leveraging interactions between FD rules, and propose methods to measure the quality of different data patterns.

- **Query-time Functional Dependency repairs.** In this work, we implement a proof-of-concept VIS that adapts our pattern-driven functional dependency repairing algorithm to work in the iterative and online settings. We show that our framework can

easily be tuned to support iterative and online data cleaning. We propose strategies to incrementally update data patterns and their quality scores as we process more query results. Our experimental results show a significant improvement in quality over a state-of-the-art rule-based data cleaning technique [16] applied at query time. The experiments also show that our algorithm is faster (by an order of magnitude) than [16] when run at query time.

Related work is discussed as we present our proposals. The rest of the dissertation is organized as follows. In Chapter 2, we present our online record linkage and fusion framework applied in the context of Web databases. In Chapter 3, we describe the fundamentals and algorithms to repair functional dependency violations. In Chapter 4, we present an adaptation of our techniques in Chapter 3 to work in the iterative and online settings. We present next directions in data cleaning in Chapter 5. Finally, we conclude in Chapter 6

2 ONLINE RECORD LINKAGE AND FUSION ON WEB DATABASES

In this chapter, we study the problem of record linkage and fusion at query time. Most existing duplicate detection and fusion techniques work in the offline setting and do not meet the online constraint. There are at least two aspects that differentiate online duplicate detection and fusion from its offline counterpart. (i) The latter assumes that the entire data is available, while the former cannot make such an assumption. (ii) Several query submissions may be required to compute the "ideal" representation of an entity in the online setting. In this chapter, we present a general framework for the online setting based on an iterative record-based caching technique.

The chapter is organized as follows. Section 2.2 describes the architecture of our system. Section 2.3 discusses caching. Section 2.4 describes the query-time answering capability of our system. Section 2.5 describes our system in action with a comprehensive example. Section 2.6 describes the experiments. Related word is in Section 2.7. Section 2.8 concludes the chapter.

2.1 Introduction

A key task in integrating data from multiple Web sources is to recognize records referring to the same real world entity—*the record linkage problem* [11]. This task is known to be difficult since the attribute values of an entity may be represented in different ways or even conflict with each other, e.g., different addresses for the same business. Conflicting data may also occur because of multiple correct values for the same real world entity, incomplete data, out-of-date data or erroneous data. Thus, another important task is to identify the correct attribute values of an entity—*the data fusion problem* [25].

In a virtual integration system (VIS), such as a vertical search engine, all of the above tasks must be performed at query time. Given that the data integration step is just one part

of the process of getting data to users (which includes network communication, ranking, etc.), it needs to be performed very *fast*. Fusing all data upfront is obviously not an option in a VIS. *The goal of this work is to provide efficient solutions to the record linkage and fusion (*RL&F*) problems at query-time.*

Table 2.1.: An example of inconsistent data on the Web.

Engine	Name	Address	Phone
Query Q1			
Metromix			
Menuism	Pizzeria Uno	49 E. Ontario St.	312-280-5115
DexKnows	Pizza Uno		312-280-5111
Yelp	Pizzeria Uno	29 E. Ohio St.	
Query Q2			
Metromix	Pizzeria Uno	49 E. Ontario St.	312-280-5115
Menuism	Pizzaria Uno	49 E. Ontario St.	312-280-5115
DexKnows	Pizza Uno	49 E. Ontario Street	312-280-5115
Yelp			

An example will help illustrate the challenges tackled in this proposal. Consider a VIS that integrates data from the following Web databases: Metromix.com, DexKnows.com, Yelp.com and Menuism.com. The following query is submitted to the VIS: $Q1 =$ (`Name` = "Pizz%"; `Cuisine` = "Pizza"; `Price` = "Affordable"; `Neighborhood` = "Downtown, Chicago"). (% is used as a wildcard character). Among all the returned records, we look at those associated with the restaurant "Pizzeria Uno" (Table 2.1). Metromix does not return any record, while the records returned by the other Web databases do not agree on the address and phone number. At this time, the best we can do is to flip a coin to decide on the address of the restaurant and to take a majority voting to decide on the correct phone number. Suppose now that at a later time a new query is posted: $Q2 =$ (`Name` = "Pizz%"; `Cuisine` = "Pizza"; `Price` = "%"; `Neighborhood` = "%"). Table 2.1 shows the set of records for $Q2$. The list of results of the two queries gives us the opportunity to see, although at different time intervals, multiple records about the entity "Pizzeria Uno". Thus, if we stored the answer to query $Q1$, then, using the answer to $Q2$ together with that to $Q1$, we could make a more informed decision about the correct address and phone

number of "Pizzeria Uno" ("49 E. Ontario St." and "312-280-5115", respectively) for any ulterior queries where "Pizzeria Uno" is a relevant answer.

The example shows that regardless of the effectiveness of the RL&F algorithms, a qualitative answer to a query Q cannot be given considering the records in response to Q alone.

Table 2.2.: Number of queries for which the overlap w/ Zagat is empty.

Web Database	# occurrences
ChicagoReader	511
YellowPages	587
Metromix	649
MenuPages	667
Yelp	673
Yahoo	676
CitySearch	702
Menuism	714
DexKnows	721

To support the above claim we conducted an empirical study where we constructed a toy VIS that connects 10 Web databases of local business listings. For efficiency purposes, the VIS collects the top-k (k = 10, 20 are commonly used) results from each Web database [26]. We submitted 1,000 randomly generated queries to each of these databases and then analyzed the overlap between the returned result lists (k = 10 was used). For example, the lists of results of Zagat and DexKnows did *not* share any record for 721 queries. As shown in Table 2.2, this was not a rare occurrence. The table shows the number of queries for which Zagat and each of the Web databases have *zero* records in common. We observe that, on average, in the merged list of results of a query, about 70% of the records appear in one or two sources (out of 10) and only about 15% of them appear in more than 5 sources. These observations clearly suggest that effective record linkage cannot in general be undertaken *query by query in isolation*, because there is not sufficient "cleaning evidence" from the records returned by a single query.

We can alleviate the lack of "cleaning evidence" by collecting "evidence" from queries as we process them. We approach the online RL&F problems from an *iterative caching*

perspective (see Figure 2.1). Specifically, the set of records corresponding to frequently posted queries is deduplicated offline and cached for future references. Newly arriving records in response to a user query are deduplicated jointly with the records in the cache, presented to users and appended to the cache. The framework of the problem addressed in this proposal is as follows:

The Setting: Let $\mathcal{D}$ be a set of Web databases. Let $\mathcal{E}$ be a set of real-world entities in the same application domain (e.g., real estate, book). Each entity has a set of attributes (e.g., name, address, phone for a business entity) and an attribute may have zero or several values. Different sources may supply different values for an attribute of an entity and the same value may be represented differently. A subset of the entities in $\mathcal{E}$ are frequently requested and a fraction of the volume of queries occurs frequently. Thus, RL&F in this environment faces unique challenges (e.g., time) and opportunities (e.g., temporal locality in queries) compared to a traditional setting.

The Problem: Let Q be a query. From the lists of records returned by the $\mathcal{D}$ Web databases in response to Q, we need to identify the set of records R referring to the same real-world entity in $\mathcal{E}$ and fuse the records in R into a single and "clean" representation. That is, solve the RL&F problems for Q.

The Proposed Solution: A solution for the above problem must achieve a trade-off between *efficiency* and *effectiveness*. Nevertheless, it needs to do so without significantly deteriorating effectiveness. We seek to improve efficiency by (1) avoiding repetitions of data cleaning steps such as unnecessary fusion, and (2) identifying the duplicates of a record in a constant time that is independent of the number of database records by using an index. Concretely, we propose an online record linkage and fusion framework (*ORLF*, for short) based on iterative caching. *ORLF* stores fused records as plain records together with information about how they were created, i.e., provenance. A non-trivial problem in the proposed RL&F framework is that of quickly finding the candidate matching records in the cache of a record in the query answer. We cannot afford to go through all the records in the cache to match them against the query records. In Section 2.4.1, we describe a

novel indexing data structure for fast similarity record lookup based on the B^{ed}-tree data structure [27].

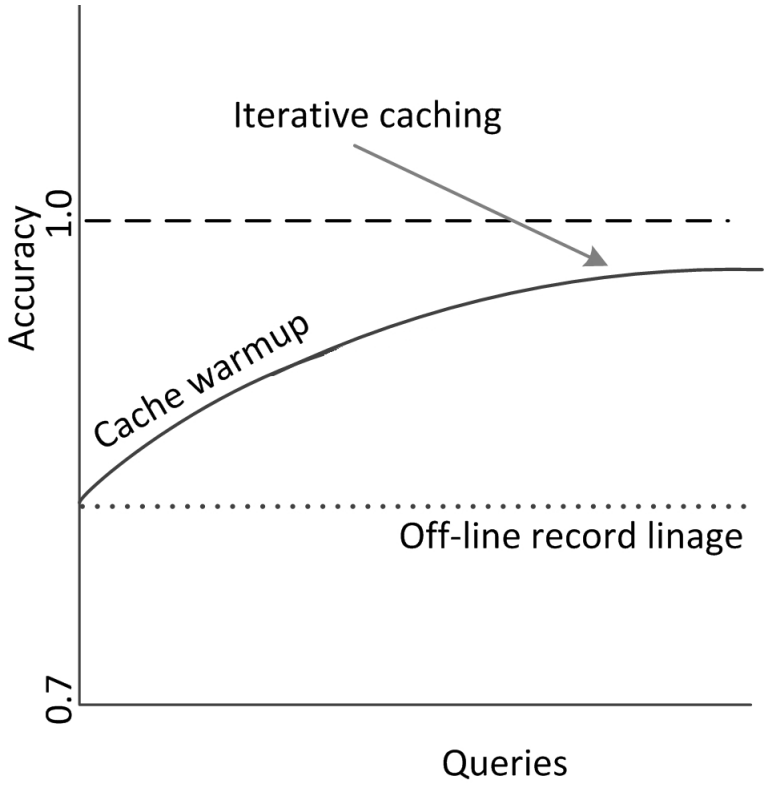


Figure 2.1.: *ORLF* versus typical RL algorithms.

Figure 2.1 is a simple sketch to illustrate our goal in *ORLF*. The Y-axis is the RL accuracy, usually reported with the classical F1-measure. The X-axis is the number of queries whose answer sets were processed. With a typical RL algorithm (aka offline in this dissertation) [28–31], the accuracy is about the same (linear) across processed queries, oscillating about their average accuracies (the dotted line). The solid line is the behavior of RL in the proposed *ORLF* framework. There is an initial warmup phase, where the accuracy is about the same as that of a typical RL algorithm. As the system processes more queries, the accuracy steadily improves. The key benefit of *ORLF* however is that the RL&F steps take milliseconds rather than seconds as with a typical RL&F algorithms. Completely building *ORLF* requires the implementation of the following components: (1) cache attribute

selection, (2) cache data structures, (3) efficient data lookup, (4) online RL&F algorithms, (5) caching policies, (6) cache warm up and (7) cache refreshing.

The contributions presented in this chapter are as follows:

- We propose, *ORLF*, an end-to-end framework to support efficient RL&F at query-time.
- We present an indexing scheme for records based on the B^{ed}-tree index [27]. B^{ed}-tree supports relatively fast record linkage and dynamic updates (for dynamic caching). To our knowledge, no other record linkage indexing technique satisfies both properties.
- We leverage query locality to perform query-time RL&F efficiently and show that smart caching avoids unnecessary fusion operations.
- We conduct extensive experiments on real and synthetic data showing the accuracy and scalability of *ORLF*.

2.2 The *ORLF* System

We describe in this section the workflow of the proposed iterative caching approach for online RL&F. In a nutshell, newly arriving records in response to a user query are cleaned jointly with the records in the cache, presented to users and appended to the cache. The workflow is drawn in Figure 2.2.

There are two processing paths: the *hit and the miss paths*. Both start by looking up the records returned by each source in response to a query in the cache. The lookup process is a two-step process. First, for each incoming record r we use an indexing data structure to retrieve the matching record of r in the cache. We employ a string-based similarity search (see Section 2.4). We collect the top-k most similar records to r in the cache. A *custom record matching function* is then used to find the matching record r in the top-k records. If a matching record, denoted r_c, exists, called a *hit*, then we take the *hit path*. If r has no

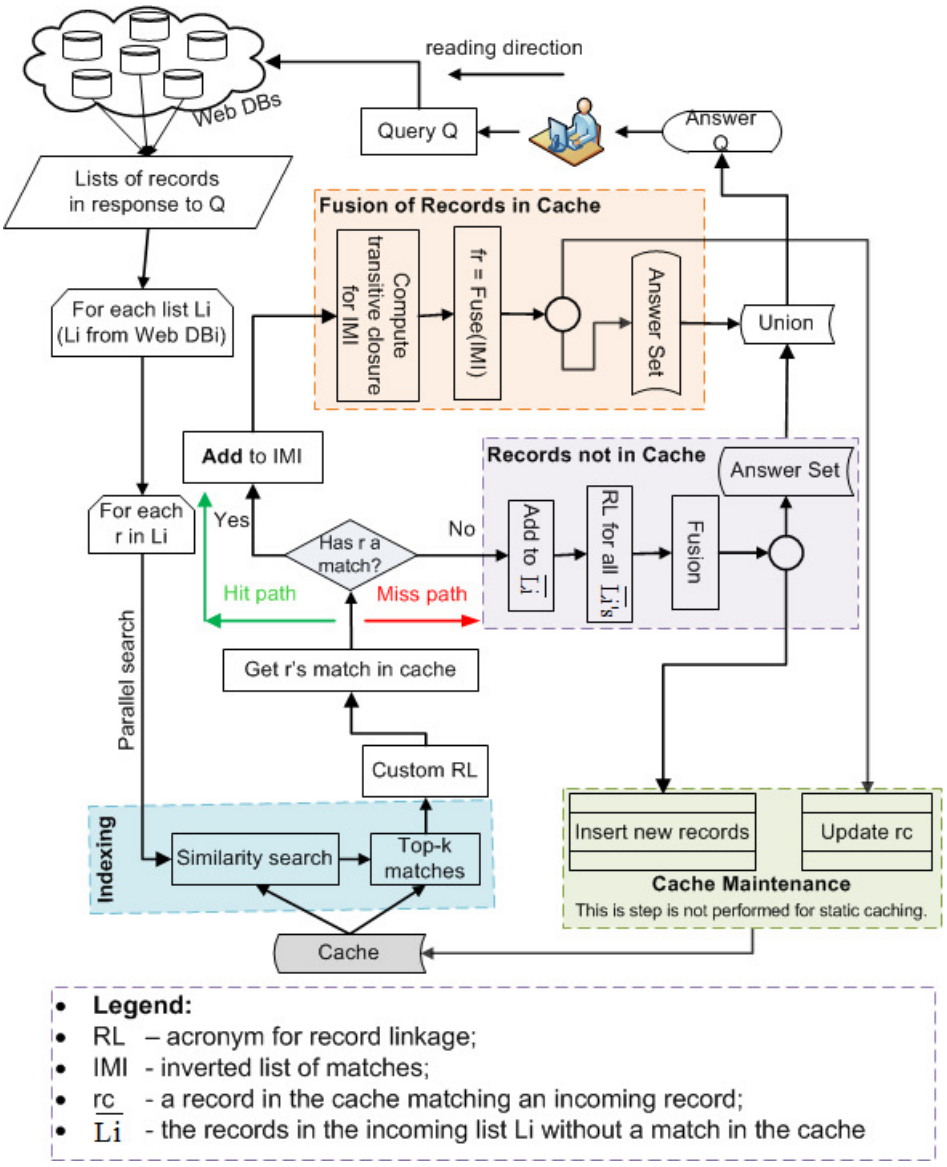


Figure 2.2.: Query answering using RL&F with iterative caching.

match in the cache, called a *miss*, we take the *miss path*. In the former, we append r_c and r to a temporary indexing data structure *IMI* (inverted match index). Each index entry in IMI is of the form $\langle r_c, LM \rangle$, where r_c is a cache record and LM is the list of incoming records matching r_c. At the end of the lookup step, IMI will contain all the record matches between the cache records and the incoming records.

Note that we do not move to the next processing block on the hit path, i.e., Fusion of Records in Cache, until we process all incoming records in response to the user query. In the Fusion of Records in Cache processing step, IMI is traversed and each r_c is fused with the incoming records in its corresponding list of matches. On the miss path, we collect all incoming records without a match in the cache and perform record matching and then fusion among them. Finally, we union the lists of records of the two paths and pass the result to the user. In practice, the union is input to a ranking algorithm (not treated in our system), which orders the records according to some user criteria (e.g., price or user rating).

The cache content is updated based on the adopted cache policy (Section 2.3). We will show that there are substantial differences between traditional caching and caching for online RL&F. Iterative caching allows a "fast" response to the current query and an "improved" data quality for subsequent queries.

2.3 Caching Mechanisms

The decision of what to cache can be taken either offline (static) or online (dynamic) in general. A static cache is based on historical information. A dynamic cache has a limited number of entries and stores items according to the sequence of requests. Upon a new request, the cache system decides whether to evict some entries in the case of a cache miss. Online decisions are based on the cache policy. Two common policies are: evicting the least recently used (LRU) or the least frequently used (LFU) items from the cache [32]. We analyze the suitability of these strategies to online RL&F. We also introduce locking of cached items and record provenance to improve efficiency when interacting with the cache.

2.3.1 Static Cache

With a static cache we need to identify the most frequently accessed entities and load their corresponding records in the cache. These entities are derived from the records corresponding to the most frequent queries gathered from the Web databases. Results of the frequent queries are processed offline using offline RL&F algorithms. We used FRIL [28]

for RL and majority voting for fusion in our prototype. More sophisticated fusion techniques can be used [33].

We introduce a variation of static caching where records in the cache are allowed to be updated, called *static cache with in-place updates* (SCU). The reason is that even though the cache content is determined offline, there can be cached records for which the number of pieces of evidence is not enough to decide whether they are correct (see example in Section 2.1). Hence, it is of practical importance to allow online updates on these records. A cache record is updated by fusing it with incoming matching records from sources which have *not* yet contributed to the cache record. We keep track of the provenance of each cache record for that purpose.

2.3.2 Dynamic Cache

The semantics of a miss is different in our caching from the traditional one. In the latter, upon receiving a request for an item v, the cache is probed for v. If v is not found, v is brought into the cache from the disk. If the cache is full, then a cached item is evicted to make space for v. In our caching setting there is no disk: all the known records are those in the cache and the ones in response to a query. Thus, on a miss, i.e., the match of an incoming record r is not in the cache, r itself is brought into the cache after RL&F. Each cache record has a *cache tag* that stores the information needed for accomplishing dynamic caching, e.g., the number of times (or the last time) the fused record was output to users.

2.3.3 Locking

Avoiding unnecessary cleaning operations is key to improving online efficiency. We use a *locking* mechanism to avoid unnecessary invocations of the fusion step. Specifically, a cache record is locked if our confidence in its quality is "high". Determining when "high" is reached for an unlocked record is orthogonal to our system. It can, for example, be probabilistic [33]. In our prototype, we implemented a voting strategy for this purpose: a record is locked if the value of each of its attributes is obtained by fusing the records from

p of the sources. We empirically set $p = \frac{|\mathcal{D}|}{2} - 1$. Note also that locking provides a means for implementing online cache refreshing: certain locked records are unlocked at certain time intervals and are refreshed via fusion. A thorough treatment of refreshing is left for future work.

2.3.4 Record Provenance

If a cached record was previously constructed using a record from a source S and, for a new query, S returns a record r that matches the fused record, then we may decide to discard r as it is very likely that r has been previously seen from source S or is a duplicate record. If the record was updated in the source, then the next refresh of the cache will reflect this change. We encode provenance using a bit string of length $|\mathcal{D}|$, such that the i^{th} bit is turned on if a record from the i^{th} source was involved in the construction of the fused record.

2.4 Query-Time Answering

In *ORLF*, the cache contains *fused records*. Specifically, if fr_i is a record in the cache at time t_i then at time t_{i+1} the new version of fr_i is either fr_i itself or a new record fr_{i+1} that is constructed out of fr_i and a set of incoming records not in the cache, which were linked to fr_i.

Algorithm 1 describes the query answering algorithm. A key novelty in our approach is that the merging is not only performed between the incoming lists of records, but the relevant records in the cache are also involved. As explained earlier, there are two processing paths: the *hit and the miss paths*. Each list of responses is processed by Procedure RecordLookupBySource (Algorithm 2) which updates the IMI (inverted match index) data structure, which will contain the set of cache records matching records in R_i and outputs Y, the set of records in R_i without a match in the cache. Y is appended to the set of unmatched records $\overline{MR}$ (Line 7, Algorithm 1). The matching records from all the incoming lists R_i and cache are fused (Line 9). The unmatched records are linked among themselves and

Algorithm 1: QueryProcessing in the *ORLF* system

Input : Query Q and Cache
Output: The set of fused records FR in response to Q and an updated cache.

```
1  Let {R_1, ..., R_m} be the lists of records from m Web databases in response to Q;
2  Initialize IMI ;                                  // the inverted match index
3  MR ← ∅ ;                                  // matched records in the cache
4  MR̄ ← ∅ ;                                  // unmatched incoming records
5  for i = 1 to m do
6  |   Y ←RecordLookupBySource(R_i, S_i, IMI);
7  |   MR̄ ← MR̄ ∪ Y;
8  MR ← Transitive closure of IMI;
9  FR_1 ←fuse(IMI);
10 FR_2 ←HierarchicalRLF(MR̄);
11 addToCache(FR_2) ;                                // only in dynamic caching
12 updateCache(FR_1) ;                              // in dynamic caching and SCU
13 return FR_1 ∪ FR_2;
```

then fused (procedure HierarchicalRLF, Line 10). The resulting fused records that do not already exist in the cache are appended to the cache (Line 11). This step is only executed for dynamic caching. Those that already exist in the cache are updated (Line 12). This step is executed when either SCU or dynamic cache are used. The union of the fused records, from both the matched and unmatched records, are then returned to the user (Line 13).

HierarchicalRLF performs pairwise RL. For example, if $\overline{R_1}, \overline{R_2}, \overline{R_3}$ and $\overline{R_4}$ are the four lists of records, then it performs RL on $\overline{R_1}$ and $\overline{R_2}$, and on $\overline{R_3}$ and $\overline{R_4}$. RL is performed again on their outputs to obtain the final list of linked records. This scheme is captured as a full binary tree, where a leaf node is the list of unmatched records from a source and an internal node represents the RL outcome on its children. While HierarchicalRLF is not as effective as a RL procedure that exhaustively compares the records of the m sources across each other, it is more amenable to a parallel implementation. Nevertheless, it is significantly more efficient and very effective within our overall framework (Section 2.6).

2.4.1 Record Look Up and Record Linkage

Let r be an incoming record and fr the record in the cache *most similar* to r. The problem is how to find fr in an online caching setting — fr may be substantially different from r, because fr has been subject to several fusion iterations. Hence, fr can only be found using similarity search. And operationally, how to check if fr indeed exists in the cache? An exhaustive search of the cache to look for fr is clearly prohibitive. We propose a two-step process: (1) fast approximate nearest-neighbor search and (2) exhaustive record matching, where we compare r with the (smaller) set of nearest neighbors (records). In light of (1), our proposed cache is an instance of *similarity caching* [34].

Nearest Neighbor Search

Given an incoming record r, we need to obtain the cached record fr such that $sim(r, fr)$ is maximized, where the similarity function is defined on the space of the "keys" of the records. A subset of the attributes of entities in $\mathcal{E}$ is a "key" if it can serve as *entity identifier*, i.e., the attributes uniquely identify an entity. For example, when matching business listings the subsets $\{\texttt{Name}, \texttt{Address}\}$ and $\{\texttt{Name}, \texttt{Phone}\}$ are such attributes. That is, if two business records have very similar names and addresses or very similar names and the same phone number, then the records are very likely to refer to the same business entity. Many indexing strategies can be used with static caching: e.g., locality sensitive hashing (LSH) [35]. For dynamic caching however, we need an indexing structure that supports efficient live updates. Since in many practical cases the entity identifier attributes are strings, we choose the B^{ed}-tree index [27], a string similarity index. B^{ed}-tree is a B^{+}-tree based index structure, which has a number of properties that suit our environment very well. B^{ed}-tree: (i) efficiently answers selection queries, (ii) can handle arbitrary edit distance thresholds, (iii) supports normalized edit distance for all query types, in particular, top-k queries, (iv) has good performance for long strings and large datasets, and (v) supports incremental updates efficiently.

Modifying the B^{ed}-tree index for use in similarity record search

B^{ed}-tree index was developed for string similarity search and hence cannot directly be used in our setting. For us, it all boils down to finding a suitable record representation such that records can be indexed with this data structure and carrying the good performance of the index on strings to records. We index a set of records using their "key" (matching) attributes $Key_r = \{A_1, A_2, ..., A_n\}$.

B^{ed}-tree index overview. B^{ed}-tree [27] is an index for string similarity search based on (normalized) edit distance built on top of a B$^+$-tree. To index the strings with a B$^+$-tree index, it is necessary to construct a mapping from the string domain to an ordered domain (e.g., the integer space). Since we are interested in top-k searches, the mapping must give an ordering that satisfies two properties when used with the edit distance: comparability and lower bounding. The former property requires linear time to verify if a string is ahead of another in the given string order, whereas the latter requires that it is efficient to find the minimal edit distance between a string q and any string in an interval $[s, s']$. The latter property is needed to efficiently prune out the intervals with no strings within the given edit distance from the query string during search. Three orderings are given in [27]: (1) dictionary, (2) gram counting, and (3) gram location. Gram counting is superior to the other two for top-k queries for relatively large strings [27]. We use it in our implementation. It is defined as follows: A string s is decomposed into a set of q-grams. A q-gram is a contiguous sequence of q characters from s. A hash function maps each q-gram to a set of L buckets. We count the number of q-grams in each bucket. At this point, s is mapped into an L-dimensional vector v of non-negative integers. The final representation of s is obtained by applying a z-order on the bit representation of the components of v. z-order interleaves the bits from all vector components in a round robin fashion. Consider the string s = "Red Lion", $n = 2$ and $L = 4$; Assuming $v = \langle 3, 2, 1, 3 \rangle$, the corresponding z-order is 11011011.

Adapting B^{ed}-tree to Record Linkage. We now describe our solution using B^{ed}-tree with the gram counting order for top-k record similarity search. A naive approach to rep-

resent Key_r is to simply concatenate the key attribute values of a record to form a string. The index key is then generated by taking the z-order representation for the obtained string. One issue with this scheme is that we may end up comparing q-grams of different attributes; e.g., if we compare two business records whose names are not of the same length, then, we will compare the q-grams of the longer name to the q-grams of the address attribute of the other record.

To avoid this problem we need to treat the attribute values as first-class citizens. We analyzed two z-order representations of record keys: *z-order concatenation* and *z-order interleaving*. In both representation schemes, we first obtain the z-order representation of each attribute. Then, in the z-order concatenation scheme the index key is obtained by concatenating the resulting bit-strings of each attribute, whereas in the latter z-order interleaving scheme, we interleave the bits from all attributes in a round robin fashion.

These representation schemes are bounded by the size of L. First, L cannot be arbitrarily large. For example, efficiency-wise $L = 4$ gives the best results in [27]. This is too small to adequately represent an index key with multiple attributes. As shown in Figure 2.5 (Section 2.6), increasing L increases the time it takes to retrieve the top-k matches. Second, for each of the n attributes we need to allocate a contiguous number of buckets. However, each attribute may require a different number of buckets. For instance, the attribute $name$ may require more buckets than the attribute $phone$. Third, we experimentally noticed that the z-order concatenation scheme is influenced by the order of the attributes, while the z-order interleaving scheme is not. So, in general we need to find a solution to the following linear equation. If we denote by L_i the number of buckets required by attribute A_i, then we have

$$L_1 + \cdots + L_n = L, \text{ where } L_i \in [b_i, B_i], b_i, B_i \in \mathbb{N}^*. \tag{2.1}$$

b_i and B_i denote the minimum and respectively maximum (ideal) number of buckets required by the i^{th} attribute. This equation may not have solutions, i.e., $b_1 + \cdots + b_n > L$. Thus, there may be application domains where B^{ed}-tree index is not suitable. Alternatively, it may have multiple solutions (this is a linear Diophantine equation). Ideally, all solutions

need to be tested out and the one that fits the application at hand the best is chosen. Enumerating all possible solutions in search for the ideal one is an overkill for some application domains because for each solution to Equation 2.1 we need to construct the corresponding B^{ed}-tree and carry out the empirical evaluation. We give here a heuristic procedure to locate a suitable solution.

We first order in descending order the attributes based on their selectivity property (selectivity of an attribute A is the ratio between the number of distinct values in A and the total number of records). This can be given by a domain expert or estimated by sampling. Intuitively, the more selective an attribute is the more useful it is to distinguish between records about different entities and thus more buckets should be allocated to it. Let $A_{i_1}, \ldots A_{i_n}$ be the desired ordering. We make $L_{i_j} = b_{i_j}$, $1 \leq j \leq n$. Then we apply a greedy strategy as follows. Let $L' = L - \sum_{j=1}^{n} b_j$. As long as $L' > 0$ we proceed as follows. For each $1 \leq j \leq n$, if $B_{i_j} - b_{i_j} \leq L'$ then $L_{i_j} = B_{i_j}$, else $L_{i_j} = b_{i_j} + L'$ and stop. For instance, suppose the attributes are name, address and phone. Suppose that this is the desired ordering and that for each of them $b = 4$ and $B = 6$. Let $L = 15$. Applying the greedy strategy, we allocate 6 buckets to name, 5 to address, and 4 to phone.

B^{ed}-tree Setup. We empirically determine that beyond $L = 12$ the performance of B^{ed}-tree deteriorates considerably. We set $b_i = 4$ and $B_i = 6$. For the matching of business listings with the attributes name, address and phone, the best configuration is to allocate each attribute 4 buckets. Also, z-order concatenation gives better retrieval times on average than z-order interleaving scheme. The order of the attributes in the z-order concatenation is name, address, phone. We further evaluate the z-order concatenation scheme in Section 2.6.1 under different parameters of the B^{ed}-tree.

Record Matching

The custom record matching function must predict with high confidence if the records r and fr refer to the same entity. Otherwise, duplicates may be inserted in both the cache and query answer. The actual function is application-dependent and, thus, orthogonal to

Algorithm 2: RecordLookupBySource

Input : (R, S, IMI): the list of incoming records R from a source S and IMI - the inverted match index

Output: $\overline{M}$ - the records in R without a match in the cache.
updated IMI.

```
1  foreach r ∈ R do
2      hasMatch ← false;
3      TK ←getTopK(r);
4      foreach fr ∈ TK do
5          if RecordMatch(r, fr) then
6              if S ∉ fr.Provenance then
7                  update(IMI, fr, r);
8              hasMatch ← true;
9              break;
10     if hasMatch = false then
11         add(r, M̄);
```

the current work. For our proof of concept, we developed such a function as follows. We obtained a training sample by posting a number of random queries to the component search engines, then we manually labeled the pairs of matching records and learnt a binary classifier: match or not match. We constructed a decision tree from the labeled data. Once we had the decision tree we transformed it into a procedure with IF-THEN rules [36] that was plugged into our system.

Look up Algorithm

Algorithm 2 describes the procedure to look up an incoming record in the cache. We first post a top-k query to the B^{ed}-tree to get the k most similar records to r in the cache. We then perform pairwise comparisons between r and the top-k records to determine fr. In our implementation, we empirically set $k = 5$ (Section 2.6). Each new record is compared against its top-k matches from the index (Lines 3-9). Our reasoning assumes: (1) cache records are distinct and (2) a cache record can match at most one record in a given source. (1) can be seen as a cache invariant and its consequence is that a new record can match at

most one record in the cache. (2) may seem restrictive, but note that our goal is not to clean the individual sources, but rather to return relevant and clean records. Thus, if r matches a cache record fr, then r is retained to be later fused with fr only if a record from S has not previously contributed to the construction of fr (Lines 6-7). If r has no match in the cache, it is appended to the list of unmatched records (Lines 10-11).

2.4.2 Fusion Procedure

For each set of records representing the same entity, we need to fuse them. For each cache attribute, the value representations that have a similarity of at least some threshold (current implementation 0.9) are considered to be "identical". Among a set of representations for a value, we choose the one provided by the largest number of sources. When a cache record fr is fused with a list of new records, the value representation of an attribute from fr receives $\lceil \frac{m}{2} \rceil$ votes, where m is the number of sources from which fr was previously derived. Other fusion schemes (e.g., [33, 37]) can easily be plugged into our framework.

2.4.3 Algorithm Complexity

The non-parallel worst-case time complexity of Algorithm 1 is $O(\frac{(m-2)(m-1)}{2}\delta(c\,R)^2)$, where m is the number of databases, each returning R records; c is the cache miss rate; $\delta \simeq 1 - or$, where or is the overlap rate of the m sources. This occurs when most of the records have no match in the cache, i.e., c is close to 1. In this case *ORLF* is simply as good as traditional RL. The best case occurs when c is close to 0. On average however, the algorithm is near linear in mR.

Table 2.3.: An example cache table

ID	Provenance	Name	Address	Phone
cr_1	(1, 2, 3, 5)	Pizzeria Uno	49 E. Ontario St.	312-280-5115
cr_2	(1, 3)	Pizzeria	40 E. Ontario St.	312-280-3344
cr_3	(1, 3)	Pizza Uno	39 N. Ontario St.	312-280-2355
cr_4	(7)	Pizzeria Uno	E. Ontario St.	312-280-5115

Table 2.4.: Answer set of a sample query

R_i	ID	Name	Address	Phone
R_1	r_1	Pizzeria Uno	49 E. Ontario St.	312-280-5115
R_2	r_2	Pizza Uno	E. Ontario St.	312-280-5115
R_3	r_3	Pizza King	11 W. Ontario St.	312-443-7844
R_4	r_4	Giordano's pizza	33 N. Ontario St.	312-544-9033

Table 2.5.: Top-2 records and their matching status for a sample query

	r_i	cr_i	Match?	edit(r_i, cr_i)
MR	r_1	cr_1	Yes	0
	r_1	cr_2	No	5
	r_2	cr_4	Yes	0
	r_2	cr_1	Yes	3
$\overline{MR}$	r_3	cr_3	No	13
	r_3	cr_2	No	13
	r_4	cr_3	No	20
	r_4	cr_2	No	20

Table 2.6.: Inverted Index (IMI) example

cr_i	Matching incoming records	Matching incoming records (after transitive closure)
cr_1	$\{r_1, r_2\}$	$\{r_1, r_2\}$
cr_4	$\{r_2\}$	$\{r_1, r_2\}$

Table 2.7.: Updated cache after processing a sample query

ID	Provenance	Name	Address	Phone
cr_1	(1, 2, 3, 5, **7**)	Pizzeria Uno	49 E. Ontario St.	312-280-5115
cr_2	(1, 3)	Pizzeria	40 E. Ontario St.	312-280-3344
cr_3	(1, 3)	Pizza Uno	39 N. Ontario St.	312-280-2355
cr_4	(3)	Pizza King	11 W. Ontario St.	312-443-7844
cr_5	(4)	Giordano's pizza	33 N. Ontario St.	312-544-9033

2.5 A Walkthrough Example

We give a step-by-step illustration of Algorithms 1 and 2 through an example in this section. Consider the following query: Q_1 = {`Address` = "%Ontario%", `Cuisine` = "Pizza"}. Table 2.3 shows the cache content. We assume four sources. Q_1 is posted to all the sources. The list of records returned from a source i is stored in a list R_i. The lists are shown in Table 2.4. For simplicity, we assume that each source returns one record in response to Q_1. For each record r in list R_i, the system finds its top-k similar records according to the string edit distance in the cache. We assume $k = 2$ in this example. Note that there is no guarantee that the returned records are indeed real matches. Hence, we next find which ones among the k records are valid matches of r. Table 2.5 shows the incoming records after the "filtering" process is applied to the top-k set. The "edit" column in the table shows the sum of the string edit distance between r_i and cr_i on attributes {name, address, phone}.

r_1 and r_2 match cr_1 and cr_4, respectively, in the cache. r_3 and r_4 have no match in the cache. The records r_1 and r_2 along with their matches cr_1 and cr_2 are processed in the "Hit path" (as shown in Figure 2.2), whereas r_3 and cr_3 are processed in the "Miss path".

Table 2.6 illustrates the usage of IMI: cr_1 matches $\{r_1, r_2\}$ and cr_4 matches $\{r_2\}$. We are assuming that the matching relation is transitive, so, by transitivity, cr_4 also matches $\{r_1, r_2\}$ after computing the transitive closure on IMI. Then, the system fuses the records cr_1, cr_4, r_1 and r_2 into cr_1. The records r_3 and r_4 do not have matches in the cache and are appended to the cache. Table 2.7 shows the new version of the cache after Q_1 is processed. The records that were either added or updated are highlighted. Finally, the system returns to the user the records: cr_1, cr_4 and cr_5.

The example shows that cache records may also be fused, such as cr_1 and cr_4. The iterative (and incremental) process gives us the opportunity to clean the cache itself of duplicates, should there be any, as more and more new queries are processed. Duplicates may sneak into the cache because there is no perfect record linkage procedure and true positives may be missed in early iterations, but discovered in later iterations.

2.6 Experimental Study

The goal of our experiments is to show that *ORLF* is feasible in practice; its effectiveness and efficiency significantly exceeds those of offline solutions, such as *Febrl* [29], when applied to the online setting. We also evaluate the main components of *ORLF* to demonstrate its robustness. All of the experiments are conducted on a machine that runs Linux, has eight Intel Xeon E5450 3.0 GHz cores and 32 GB of physical memory. We implemented the framework in C++ and used MySQL to manage the data in the sources and the cache.

2.6.1 B^{ed}-tree Index Experiments

We assess the effectiveness of the modified B^{ed}-tree index in the record linkage task. The effectiveness of B^{ed}-tree is not analyzed in [27].

We define the *sensitivity* of the modified B^{ed}-tree index as its ability to return a record r in response to the top-k query r given that r is present in the index. If the index has a low sensitivity, then *ORLF* does not benefit from caching because it cannot locate the matching records of the incoming records even when they are in the cache. We assess the sensitivity of the B^{ed}-tree index with the two strategies of representing the matching key, naive and z-order concatenation.

The sensitivity experiment requires a set T of N distinct records w.r.t the record matching function discussed in Section 2.4. We randomly generated over 1M records for the attributes (name, address, phone) with an approximate error rate of 20%, i.e., around 20% of the records have duplicates in the set. This does not bias the experiments as we are interested in measuring the sensitivity of the B^{ed}-tree when *most* of the records are distinct– Recall that the cache is assumed to be duplicate-free in general. We have also conducted an experiment where we indexed a set T_D of 1K records that are guaranteed to be 100% distinct from each other.

The experiment runs as follows: (1) records in the set T are indexed using the modified B^{ed}-tree index ; (2) a sample set of records, ST, of size 100 is taken from T; (3) a top-k

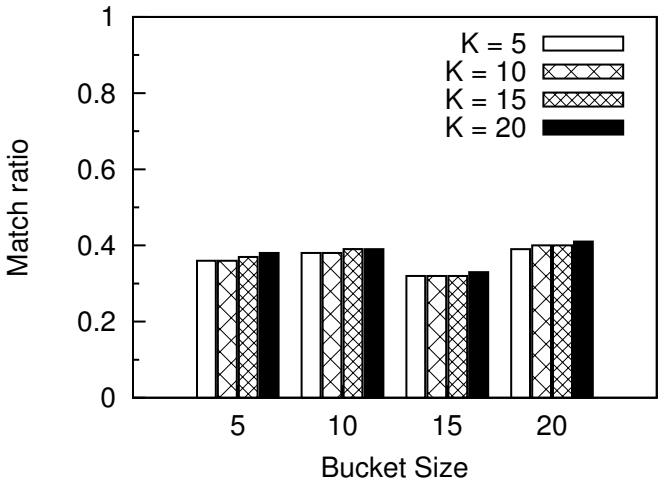


Figure 2.3.: Matching ratios using different values of k and L (strings concatenation)

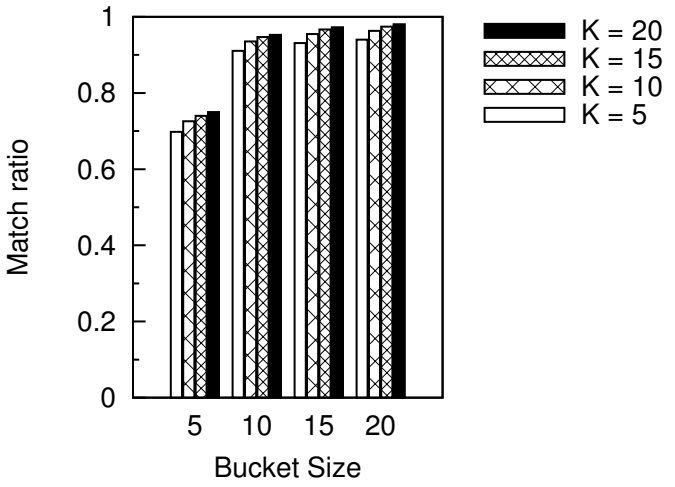


Figure 2.4.: Matching ratios using different values of k and L (z-order concatenation)

query to the B^{ed}-tree with every record r in ST is posted; and (4) if $r \in T_r$, where T_r is the set of returned records for r, then it is a match; otherwise, it is a miss.

When the indexing key of a record is obtained by mere concatenation of its attribute values, B^{ed}-tree exhibits a poor quality as illustrated in Figure 2.3. The main reason is that attribute values have different lengths, thus different parts of these attribute values are compared when computing the lower-bound of the edit distance.

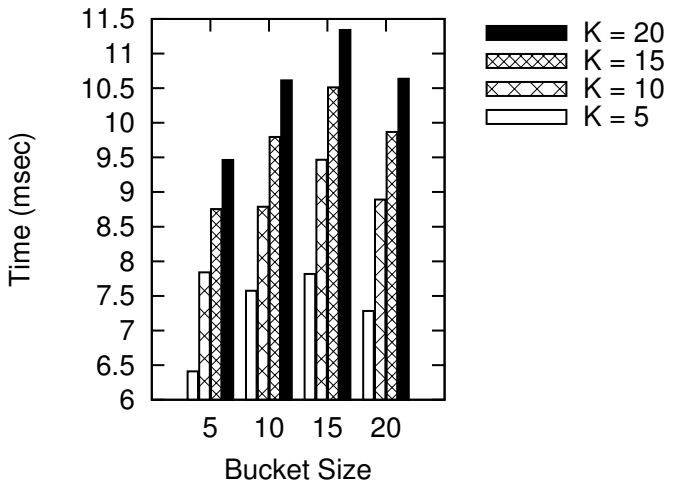


Figure 2.5.: Average Top-k query time using z-order concatenation to index records

Figure 2.4 shows the matching ratios when the indexing key is the z-order concatenation of the attribute values of a record. The matching ratios are reported for different values of k and L (the bucket size). The matching ratios are reasonable and comparable to those for the original B^{ed}-tree index. This z-order concatenation strategy is used in our implementation and all the subsequent experiments. In another set of experiments, we indexed T_D and then queried the index with all of the records in T_D. We obtained the average matching ratios 0.97, 0.98, 0.99, 0.99 for k = 5, 10, 15 and 20, respectively. This shows that the z-order concatenation scheme is very effective and suitable for indexing records with the B^{ed}-tree index.

While increasing L and k has a clear positive impact on the match ratio as shown in Figure 2.4, Figure 2.5 shows a negative impact on efficiency. As a reasonable trade-off between quality and performance, we set the overall bucket size L = 12 with 4 buckets per attribute, and k = 5.

B^{ed}-tree Effectiveness for Plain Strings

We also need to have a sense of the effectiveness of the B^{ed}-tree index in general. We use the original implementation of the B^{ed}-tree[1]. We employ a set of 12K distinct strings corresponding to business names and post top-k queries. On average, we obtained a 0.99 matching ratio for k = 5, 10, 15 and 20. Larger k values are not used in practical systems in the online setting. The B^{ed}-tree index is indeed effective when applied to plain strings. This will serve as the baseline behavior for the modified version of B^{ed}-tree.

2.6.2 *ORLF* Experiments

We implemented an in-house metasearch engine to create a controlled experimental environment, we used two datasets: (1) Real Web data, to show how the system behaves in the wild; (2) Synthetic data, to show that the system is not sensitive to the data domain being used, and to better evaluate the RL quality since we know the set of duplicates for all the records. To obtain the Web dataset, we crawled the data of 9 Web databases that provide information about restaurant listings in the metropolitan Chicago, US. We also compared *ORLF*'s fusion with *Solaris* [37] on a book dataset.

Table 2.8.: Dirtiness statistics of the crawled data

		Number of records with distinct				**Number of records without**								
Source	# Recs.	**Name**	**Address**	**Zip**	**Phone**	**Name**	**Address**	**City**	**State**	**Zip**	**Phone**	**Rating**	**Reviews**	**Price**
ChicagoReader	3,096	2,759	2,877	0	2,930	0	0	0	0	3,096	35	1,436	1,436	209
CitySearch	12,695	9,162	9,867	124	0	0	0	0	0	3,035	12,695	8,883	6,866	12,695
DexKnows	5,843	4,577	5,509	61	5,706	0	0	0	0	3	3	5,636	5,636	5,843
MenuIsm	8,508	6,492	7,022	0	0	0	0	0	0	8,508	8,508	6,795	6,795	889
MenuPages	3,629	3,032	3,382	0	0	0	0	0	0	3,629	3,629	1,465	1,464	3,629
Metromix	5,044	4,599	4,719	0	0	0	7	7	0	5,044	5,044	1,627	1,627	5,044
Yahoo	10,820	8,049	9,724	0	10,490	0	47	0	0	10,820	0	5,854	5,854	10,820
YellowPages	7,798	6,547	7,159	101	7,485	0	21	1295	0	17	0	4,741	4,741	7,798
Yelp	10,115	8,200	8,914	107	8,744	0	0	87	0	37	367	2,080	2080	255

Table 2.8 gives a general picture of the "dirtiness" of the data crawled from the 9 Web databases. The part titled "Number of records with distinct" includes four of the cache

[1]We thank the authors of [27] for sharing their source code.

attributes. "City" and "State" are omitted as most records are from Chicago, IL. We show the number of distinct values in each of the cache attributes, e.g., there are 8,200 distinct names in Yelp. A "0" in the column of an attribute means that there are no values for the attribute in the corresponding crawled data, e.g., no record has a zip code in the crawled data from MenuPages. The part of the table titled "Number of records without" looks at the missing values in all the attributes. If we analyze the two tables jointly, we note that all the records have a name value, but not all of them have a phone number, e.g., in Yelp there are 8,744 distinct phone numbers and 367 records have no phone number.

A key question is whether the content of real Web databases overlaps significantly. We use *overlap rate* to measure the degree of overlap among a group of databases [38]: $\frac{\sum_{i=1}^{m} |D_i| - |D_u|}{(n-1)|D_u|}$, where m is the number of databases, $|D_i|$ is the number of records in database D_i and $|D_u|$ is the total number of distinct records in the union of the n databases. The overlap rate for our metasearch engine is 0.38. This is a global score; pairwise, the databases have higher rates. Consequently, the likelihood of at least two records referring to the same entity to occur in a query is very high. This emphasizes the practical importance of addressing the problem of online RL&F.

Matching Function Thresholds: We use the dataset 12Q[2] for learning and the Restaurant dataset from the RIDDLE[3] repository for testing. We obtain 12Q by posting queries to the 9 component search engines. The thresholds to match restaurant entities have been learned and applied to the *ORLF* system custom function to match restaurant pairs from the real-world data crawled from the Web.

Cache Warmup: A well-known result from information retrieval is that query frequencies follow a power-law distribution for text search engines [39, 40]. Thus, a few queries have very high frequencies and the rest appears very infrequently. To our knowledge, there is no similar study for the queries posted over search engines for (semi)structured data. It is however known that the sales of books, music recordings and almost every other branded commodity follow a power law distribution [41, 42]. Hence, we can "deduce" that the queries used to find them also follow a power law distribution. To warm up the cache, we

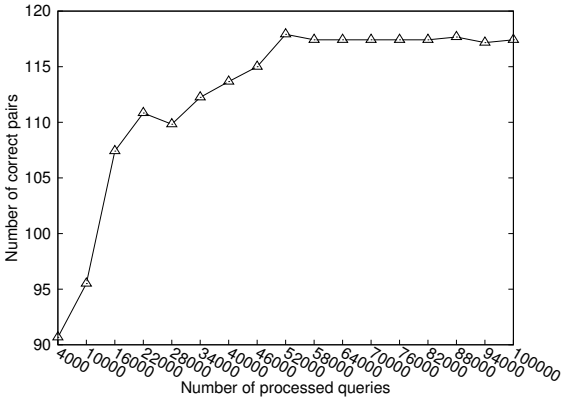
[2] `www.cis.temple.edu/~edragut/research.htm`
[3] `www.cs.utexas.edu/users/ml/riddle/data.html`

generate a stream of structured queries following a power law distribution. We select the top 20% most frequent queries, post them, and download their results. We then apply an offline RL&F algorithm, namely FRIL [28], on their results.

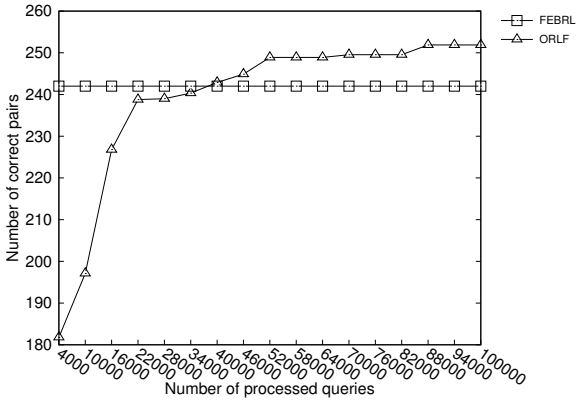
Generating Simulated Queries: To empirically analyze the behavior of *ORLF* we need to generate queries that simulate the influx of queries faced by a search engine. To our knowledge, there is no published method for simulating a *stream of structured queries* to a search engine. We give a method here. First, suppose that we know the set of query fields, say F. For example, $F = \{\text{Cuisine, Price, Location}\}$ are common fields in restaurant search engines. We assume that each field f has a predefined list of values D_f. This is quite common on the Web. Otherwise, we can draw values from some sample datasets. We then generate the entire query space by taking the cross-product of the domains of the fields, $\prod_{f \in F} D_f$. We also insert a null value in the domain of f in order to account for the queries when f is not mentioned. For instance, the query (`Cuisine` = "Mexican"; `Neighborhood` = "Loop, Chicago") does not mention the price. We draw a stream of queries from the set of all queries according to a power law distribution, i.e., $p(x) = C\, x^{\beta}$. $C = \frac{\beta+1}{Q^{\beta}}$, Q is the total number of distinct queries. We set β to values observed in literature (e.g., [40]) for keyword queries, e.g., $\beta \in \{0.83, 1.06\}$. We denote the generated stream of queries by Q.

Obtaining the Gold Standard: Since no automatic RL tool can guarantee perfect results, we manually construct a subset of matching records. We randomly select a subset R_M of 100 records from the crawled data. Then, we apply the record linkage tool *Febrl* to R_M and the entire crawled data and obtain a set of candidate matches for the records in R_M. We manually investigate the generated pairs to keep only the correct matching pairs P_{GS}. P_{GS} contains 420 pairs. We use P_{GS} to measure the effectiveness of *ORLF*.

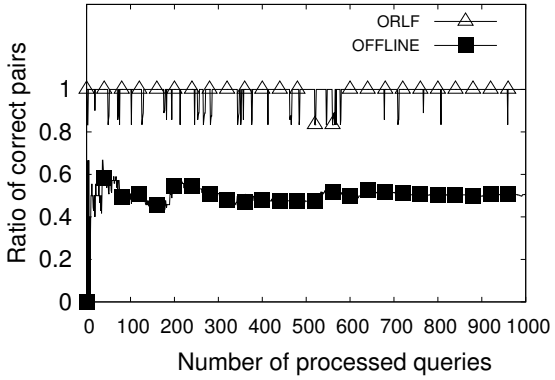
Dynamic Cache with Infinite Size: In these experiments, *ORLF* is set up with a dynamic cache of infinite size, and k=5, L=12 for the B^{ed}-tree index. In the first part, we evaluate the quality of *ORLF* in the task of RL against P_{GS}. We simply count correct matching pairs that *ORLF* returns as it goes through the stream of queries. The experiment is conducted in the following manner:



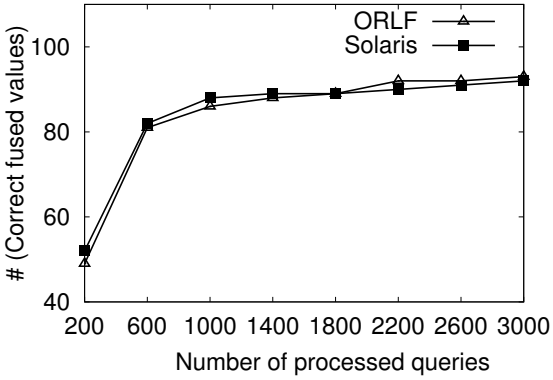
(a) *ORLF* vs. Gold Standard



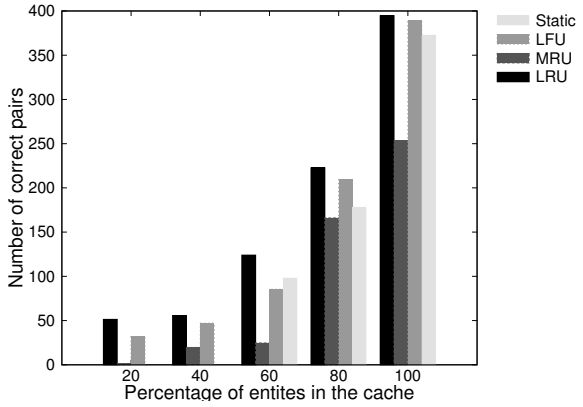
(b) *ORLF* vs. *Febrl*



(c) *ORLF* vs. offline



(d) *ORLF* vs. *Solaris* (effectiveness)



(e) Different cache configuration policies

Figure 2.6.: *ORLF* quality experiments

1. Let Q be a stream of queries and $Q_M \subset Q$ a sub-stream of queries with the property that the list of results of each query contains exactly one record in R_M and each record in R_M appears in the list of results of some query.

2. We run 1,000 queries from $Q - Q_M$. Then, we run all queries in Q_M randomly.

3. For each query $q \in Q_M$, *ORLF* yields a set of duplicate record pairs P_q corresponding to q.

4. We collect all matching pairs P_{ORLF} generated by *ORLF* (Eq. 2.2) over the entire stream Q_M. Then, we extract the set of correct matching pairs from P_{ORLF} (Eq. 2.3) .

$$P_{ORLF} = \cup_{q \in Q_M} P_q \tag{2.2}$$

$$Correctness_{ORLF} = |P_{ORLF} \cap P_{GS}| \tag{2.3}$$

5. We repeat 2-4 until *ORLF* processes 100,000 queries. The goal is to show that *ORLF* incrementally benefits from past queries and yields significantly improved RL results.

In general, both P_q and P_{ORLF} are different at subsequent iterations because more and more records are appended to the cache. $Correctness_{\text{ORLF}}$ is independent of the number of processed records when performing RL. Hence, it is a good indicator of *ORLF* effectiveness since it provides a uniform way to measure effectiveness across iterations. The goal is to show that *ORLF* converges to P_{GS} as the system processes more queries. Note that in this and in the following experiment, to increase the randomness of the testing, we tested *ORLF* with three different query streams, corresponding to different values for the parameter $\beta \in \{0.64, 0.7, 1.3\}$ of the power law distribution, and reported the average number of correct pairs across the three runs.

Figure 2.6a shows that the overall quality of *ORLF* improves sharply as the system processes more queries, then it remains relatively stable. More importantly, it does not deteriorate; *ORLF* benefits from previously processed queries and improves the quality of the current and future queries.

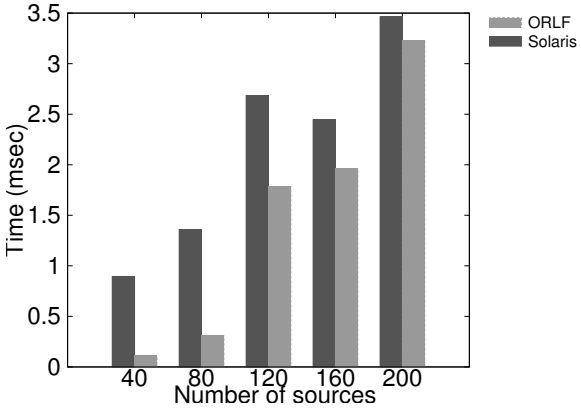
The second part of the experiment compares our system to simply applying an off-the-shelf offline RL tool. We choose *Febrl* [29] because its source code is readily available

online. *Febrl* takes two sets of records S_1 and S_2 as input and outputs the set P_{Febrl} of duplicate pairs from these two sets. We employ the same setting described above. *Febrl* is used as follows. For each query $q \in Q_M$ let $D_i(q), 1 \leq i \leq 9$ be the set of results returned by the i^{th} Web database in response to q. *Febrl* is then applied to every pair $D_i(q)$ and $D_j(q)$ of result sets, $1 \leq i < j \leq 9$. *Febrl* is applied to 36 pairs of result sets per query. (This experiment is the most time consuming and it took several days to complete). We then union the pairs of matching records obtained from the 36 runs of *Febrl*, we call this set $P_{Febrl}(q)$. The set of pairs of record matchings produced by *Febrl* for all the queries in Q_M is given by $P_{Febrl} = \bigcup_{q \in Q_M} P_{Febrl}(q)$. P_{Febrl} is computed every 1,000 queries as is P_{ORLF}. The set of correct pairs of matching records for *Febrl* is given by $Correctness_{Febrl} = |P_{Febrl} \cap P_{GS}|$.

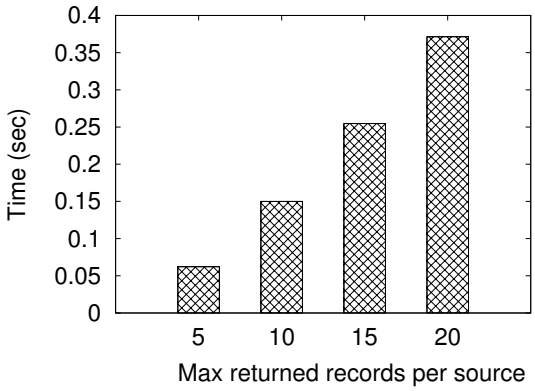
Figure 2.6b plots $Correctness_{Febrl}$ and $Correctness_{ORLF}$ side by side. The graph clearly shows that for the initial set of queries, *Febrl* outperforms *ORLF*. However, as more queries are processed, *ORLF* starts to gradually catch up with *Febrl* and eventually outperforms it. Figure 2.6b approximately mirrors the trend presented in Figure 2.1. Observe that *Febrl*'s effectiveness remains about the same across the query stream.

Dynamic Cache with Eviction and Static Cache: We evaluated two caching strategies: dynamic caching with three eviction policies (LRU, MRU and LFU) and static caching. We use *Febrl* to clean all the crawled data from the 9 databases offline; the resulting clusters (or entities) are loaded increasingly by fraction into the cache. We report the number of returned correct pairs (Eq. 2.3) computed from randomly selected queries from Q. We report the numbers for each of the considered cache sizes.

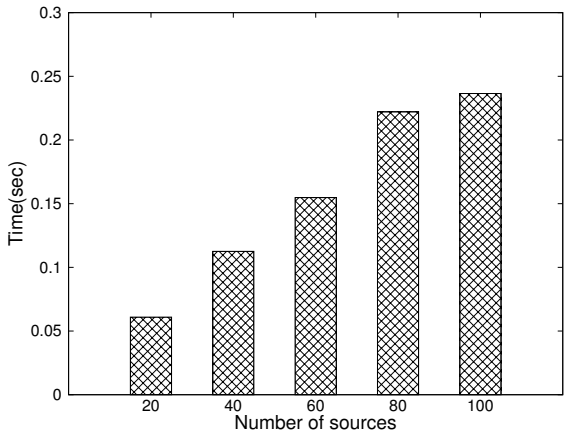
Overall, LRU has the best results and MRU has the poorest as shown in Figure 2.6e– MRU evicts the most recently used entity from the cache, which is likely to be needed for future queries. The static cache performs worse than the dynamic cache, but it becomes almost as good as the dynamic cache as the cache size approaches 100% of the total entities. LRU and LFU dynamic caches show better peformance than the static cache since a dynamic cache continues to update the content of the cache over time, while a static cache



(a) *ORLF* vs. *Solaris* (average response time)



(b) Average response time



(c) Scalability *w.r.t.* to the number of sources

Figure 2.7.: *ORLF* average response time

does not. Increasing the cache capacity allows *ORLF* to evict less records (and thus, to keep more "useful data"); this decreases the likelihood to miss an incoming record.

We report experiments that assess *ORLF*'s performance. First, we evaluate the average query response time by varying the number of returned records per source, we used a total of 10 sources in this experiment. Figure 2.7b shows the average query response time when using different values for the maximum number of returned records per source, we can see that *ORLF* is very efficient, even when processing 20 records from each source, the average query response time does not exceed 0.4 second. We also evaluate the scalability of *ORLF* by computing the average query response time for a set of 200 randomly selected queries,

while increasing the number of sources to which we submit the queries. The response time is the time between querying the cache for top-k matches and producing the query's output. We do not take the database querying time into account as it depends on factors (e.g., access method, internet bandwidth) that are outside the scope of this work. We start with 9 distinct sources. We gradually increase the number of sources up to 100 by duplicating the original sources. Figure 2.7c shows the scalability results. We observe that the increase in the query response time is linear in the number of sources. We have observed that the query response time is primarily dominated by the top-k query time, this is expected because the top-k queries in the B^{ed}-tree index require computing edit distance for each string in the tree leaves to find the possible matches to the querying record. In addition, the similarity search algorithms based on B-tree indices explore many branches (in the worst case all of them) in the tree that may contain candidate matches [43].

2.6.3 Synthetic Dataset

In this experimental study, we compare *ORLF* against an ideal system that has a perfect RL algorithm. That is, the ideal system does not miss a pair of matches in the list of returned records of a query. We want to showcase that because the system is stateless (does not keep information from previous queries), it cannot deliver a "clean" representation of an entity in most cases, while *ORLF* does. The experiment is set up as follows. For an entity e, let L_e be the set of records from all sources that need to be known for a fusion algorithm to compute the correct representation of e. L_e is computed by RL (over time). Missing any portion of L_e would render an imperfect version of e regardless of the fusion algorithm being used. The "cleanliness" of e is measured as the ratio $|L|/|L_e|$, where L is the set of records discovered by one of the two systems. For *ORLF*, L is the union of the pairs discovered across processed queries, while for the perfect offline RL system, it is the set of matching record pairs for a given query.

We use synthetically generated data to track the duplicates of all the records in the dataset, and hence, to accurately measure the effectiveness of the two systems. We gen-

erated a synthetic dataset of 1M records that contains made-up personal information such as first name, last name, and social security number. We used the tool *dsgen* which is part of *Febrl* to generate this dataset. We generated 200K records along with 800K duplicates. The maximum number of duplicates per record was set to 10. The tool introduces different types of noise to the original records to generate duplicates. In order to simulate the setup of a VIS, we randomly split the set of synthetic data into 10 sources, each containing 100K records.

The matching function checks if two records have a similar SSN, surname and phone. The string edit distance threshold for the three attributes is empirically set to 0.8.

Figure 2.6c shows the results obtained for a query stream of 1K queries posted to 10 sources; we can see that *ORLF* greatly improves the quality of the returned results to the posted queries over the perfect RL tool. Such improvement comes from the proposed caching system which makes *ORLF* benefit from previously processed queries.

2.6.4 Comparison with *Solaris*

We compare *ORLF* to *Solaris* [37], which performs online fusion of records returned by a query with copying and accuracy constraints on the sources. *Solaris* has two methods: *ACCU* and *PRAGMATIC*. As reported in [37], *PRAGMATIC* has a slightly better precision than *ACCU*, but it is slower. We only implemented *ACCU* for the comparison.

Dataset: We use the dataset *Books* [37][4] since *Solaris* requires the accuracy information of the Web sources. *Books* has book records (ISBN, title and authors) from 894 sources. It comes with a gold standard set of 100 books for which we know the correct authors.

Query stream: We generate a set Q_{books} of 3000 queries that ask for the books in the gold data. For a book entity B_i that has a set of records $R_i = \{r_1^i, r_2^i, ..., r_n^i\}$ across the sources, there exists a set of queries that have different coverage ratios on R_i. $ORLF$ is expected to do well even when the query coverage ratio is low (thanks to the cache), whereas *Solaris*,

[4]We thank the authors for sharing with us the dataset, the accuracy of the sources, and the gold standard.

due to its stateless nature, performs well when the query has a high coverage ratio. We consider queries of different coverage ratios to be fair to both systems.

We post the queries in Q_{books} to *ORLF* and *Solaris* to assess the accuracy of their fusion results. We measure this accuracy by assessing their ability to return the correct value for the authors of the books in the gold data. We consider only the sources that contain books in the gold data. There are 238 such sources. As we see in Figure 2.6d, the two systems perform quite similarly. However, unlike *Solaris*, $ORLF$ performs RL (besides data fusion). $ORLF$ is thus exposed to RL mistakes (which may lead to low-quality fusion). Data fusion in the current implementation of $ORLF$ is naive and only considers majority voting among conflicting values. We observe that in the beginning, *Solaris* slightly outperforms $ORLF$. As $ORLF$ processes more queries, it starts outperforming *Solaris* as it benefits from the cached fused results.

Using the same dataset, we compare the average response time for both systems, excluding the data sources querying time. We use a query stream of 100 randomly chosen queries from Q_{books}. Figure 2.7a shows that *ORLF* slightly outperforms *Solaris* since it does not perform any preprocessing for the conflicting values to be fused, it just takes the majority value. As the number of data sources increases, *Solaris*'s average time becomes closer to *ORLF*'s; this is because *Solaris* does not necessarily query all the data sources, and hence it is not as sensitive to the number of sources as *ORLF*.

2.7 Related Work

Our work is related to four areas: metasearch engines for structured data, data fusion, record linkage and caching. It also builds upon existing offline RL&F techniques, most of them comprehensively summarized in [11] and [44]. We are not aware of any caching method for metasearch engines over structured data. Caching nonetheless has received substantial consideration in text Web search engines [45].

To our knowledge, an RL&F system for online settings as presented in this chapter, i.e., for metasearching, has not been proposed before. We are aware of three other recent works

that propose online solutions [33, 37, 46]. In [46], the term "online" denotes an entirely different setting than ours, namely, a set of distributed databases whose records need to be matched over a network. The goal is to minimize the communication overhead, i.e., to minimize the number of records transferred over the network.

The approach proposed in [37] (referred to as *Solaris*) performs online fusion with copying and accuracy constraints on the data sources. The key idea is to stop probing additional sources, in response to a query, once the system is confident enough that data from the remaining sources are unlikely to change the answer computed from the probed sources. The main differences between *Solaris* and our system are: (1) *Solaris* does not perform RL (in the reported experiments, ISBN is assumed clean and used as key for RL purposes) and (2) *Solaris* is stateless, thus its fusion accuracy for an attribute value is as good as the number of records having that value or related values in the query result set. Our experiments on the same dataset show similar accuracy for both systems.

A Bayesian approach for fusing records is also described in [33]. It infers the quality of a data source for different attribute types without any supervision and incorporates this quality in the fusion process. While both works present interesting frameworks for online fusion, there are at least two issues with these solutions. First, as illustrated in Section 2.1, the lists of returned records for a query do not a have high degree of overlap. Second, as shown in these works, a large number of sources may need to be probed to reach the desired quality level at query-time. For example, in [37] the authors show experimentally that 73 out of 100 (book) records reach a stable version after 14 sources are probed and all 100 are stable after over 90 sources are probed. Although a metasearch engine may connect to hundreds of component search engines, in practice and for efficiency reasons, it submits a query to a very small number of them: a few tens of them [47]. The databases are selected based on the query at hand and the profile of each database [26,48]. The above two observations suggest that the proposed fusion approach may not be suitable for metasearch engines. Data from past queries, i.e., caching, is needed for accurate online data cleaning.

While iterative caching was not used to efficiently perform data quality, efficient (offline) RL nonetheless is a major topic. Methods to speed up the performance of RL include

iterative blocking [30], size filtering [49], order filtering [50], suffix filtering [51], iterative hashing [31] or "hints" as in the pay-as-you-go technique proposed in [52]. Three type of hints are proposed: sorted lists of record pairs, partition hierarchy, and sorted list of records.

Incremental record linkage is also a related topic [21,53,54]. An *incremental clustering* technique is proposed in [53]. For a new record, it estimates its likely cluster through a voting scheme and then it recursively updates the clusters. The algorithm is not suitable for online settings because the recursion may pass through the entire database. The solution in [54] is a heuristic incremental clustering algorithm that ignores the propagation step. The neighboring objects are never analyzed, the clusters never merge or split: a new record is either added to a cluster or forms a new cluster. [21] proposes two graph incremental clustering algorithms for RL. The proposed algorithms are intractable in general and the proposed solutions are not suitable for the online setting.

[55] reports a probabilistic approach to online RL. The aim is to return alternative linkage assignments with assigned probabilities in a response to a query, whereas we return duplicate-free query results along with fused attribute values.

2.8 Concluding Remarks

In this chapter, we presented a novel approach for record linkage and fusion in an online setting. Our approach is based on *iterative caching*: a set of frequently requested records (obtained from the different Web databases through sampling) is cleaned offline and cached for future references. Newly arriving records in response to a query are cleaned jointly with the records in the cache, presented to users and appropriately appended to the cache. Our solution allows a "fast" response to the current query and an "improved" data quality for subsequent queries.

There are at least two items for future work: (1) Devise a measure of degradation of the cache, which would trigger a cache refresh. (2) Incorporate better fusion algorithms in

ORLF. The solution presented in [33] seems to be better amenable to our framework due to its incremental nature.

3 PATTERN-DRIVEN DATA CLEANING

A large class of data repair algorithms rely on data-quality rules and integrity constraints to detect and repair the data. A well-studied class of integrity constraints is Functional Dependencies (FDs, for short) that specify dependencies among attributes in a relation. In this chapter, we address three major challenges in data repairing: (1) Accuracy: Most existing techniques strive to produce repairs that minimize changes to the data. However, this process may produce incorrect combinations of attribute values (or patterns). In this work, we formalize the interaction of FD-induced patterns and select repairs that result in preserving frequent patterns found in the original data. This has the potential to yield a better repair quality both in terms of precision and recall. (2) Interpretability of repairs: Current data repair algorithms produce repairs in the form of data updates that are not necessarily understandable. This makes it hard to debug repair decisions and trace the chain of steps that produced them. To this end, we define a new formalism to declaratively express repairs that are easy for users to reason about. (3) Scalability: We propose a linear-time algorithm to compute repairs that outperforms state-of-the-art FD repairing algorithms by orders of magnitude in repair time.

The chapter is organized as follows. We present the preliminaries in Section 3.2. We define the space of repairs and the problem in Section 3.3. Section 3.4 presents the formalisms we use to express repairs in terms of their underlying FD patterns. Section 3.5 presents the metrics we propose to compute the quality of FD patterns. In Section 3.6, we present the building blocks to our repairing technique and propose a set of data repairing algorithms. We highlight key experimental results in Section 3.7 and review related work in Section 3.8. Finally, Section 3.9 concludes the chapter.

3.1 Introduction

In rule-based data cleaning, various types of rules have been proposed to characterize clean data including functional dependencies (FDs) [17], conditional FDs (CFDs) [56], inclusion dependencies [17], and denial constraints [16]. While the ultimate goal of data cleaning is to take a data instance from its "dirty" state to its "clean" state, i.e., the ground truth, most automatic rule-based data-repairing tools only guarantee consistency of the data with respect to the defined rules. This process may not necessarily lead to the "truth" version of the data.

In general, correct data is a genuine representation of reality. Hence, correct values will maintain some data patterns based on their distribution and relationships to each other [57, 58]. For example, in Figure 3.1, the pattern [country = "Germany", capital = "Berlin"] is strongly supported in the data compared to the pattern [country = "Russia", capital = "Berlin"]. In a situation where we need to change the data (or repair it) due to some errors, these changes or repairs should strive to *keep* data patterns that are likely correct. In the example, we would strive to keep the former pattern. The main focus of current repairing algorithms is to compute repairs that minimize the changes to the data *without* considering the overall effect of each repair on the underlying data patterns. We illustrate this limitation through a motivating example.

Example 3.1.1 *Consider Table Tour in Figure 3.1 listing names and countries of cyclists that participated in Tour de France 2016. Consider the following FDs defined over Table Tour:* $fd_1 : cyclist \rightarrow country$ *and* $fd_2 : country \rightarrow capital$. fd_1 *states that records with the same cyclist name must have the same country while* fd_2 *states that records with the same country name must have the same capital.*

Standard solution. Tuples t_1 and t_2 violate fd_1. To repair this violation, most data-repairing algorithms would change the value of any of the four cells involved in the violation. For example, by changing the value of $country$ to either "Germany" or "Russia", the violation will be eliminated and the data instance will be consistent with the two FDs. At a first glance, both values seem equally good because they appear once for the cyclist

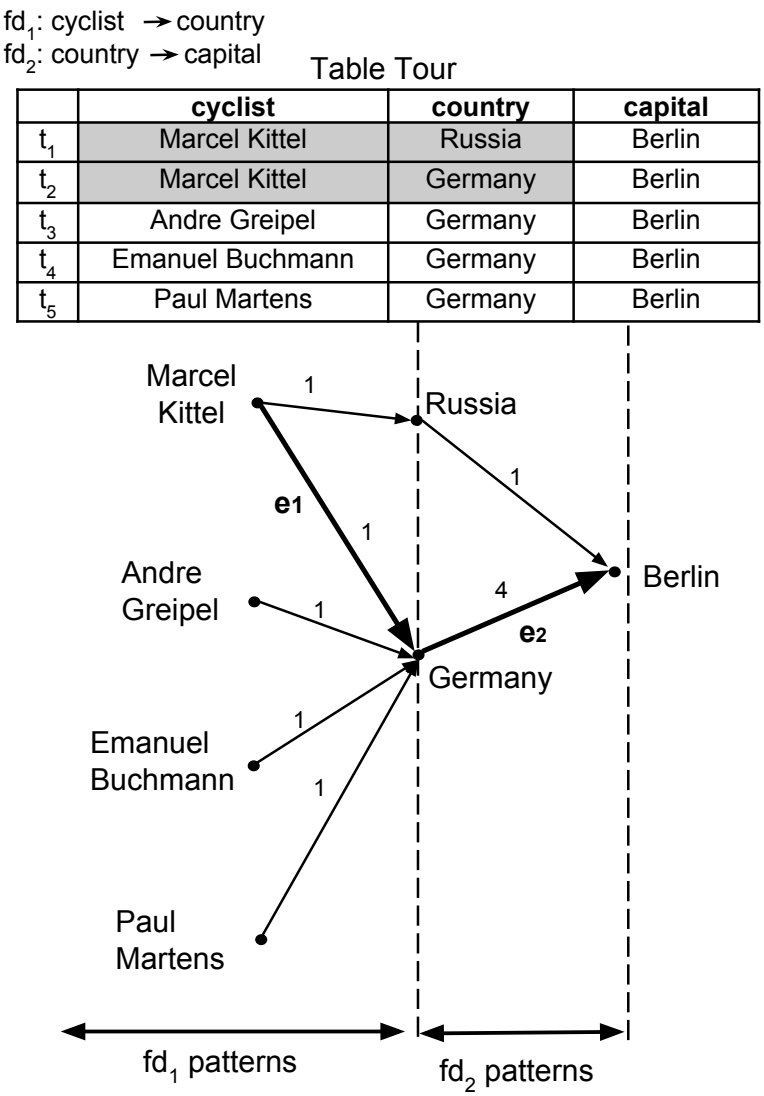


Figure 3.1.: Sample instance and the graph representing data dependencies with respect to fd_1 and fd_2

"Marcel Kittel". Choosing "Russia" would result in a consistent instance but would create *incorrect* combinations with values from other attributes. For instance, the value combination [country="Russia", capital="Berlin"] is incorrect. Existing repairing algorithms look at the violating values in isolation from the non-violating ones. Consider the example above. Looking at the values "Russia" or "Germany" in isolation from other attribute

values may create *incorrect* value combinations with other attribute values. Therefore, it is important to recognize and preserve correct value combinations across multiple attributes.

Modeling Value Combinations. One way to capture value combinations that bind semantically-related attributes is through FDs. When instantiated on the data, these dependencies form data patterns that bind together semantically-related data values. For instance, the pattern [country = "Germany", capital = "Berlin"] is a binding of data values [country = "Germany"] and [capital = "Berlin"] through fd_2. Consequently, every FD generates a set of patterns. We refer to these patterns as *FD patterns*. In our proposal, we treat FD patterns as first-class citizens. We extract FD patterns from the dirty data and reason about their quality and interactions to compute a repair. The goal is to add more context to different repair choices by looking at data values as members of data patterns. Thus, updating a value would update its underlying patterns. In other words, we want to ensure that the introduced repairs maintain data patterns that are most likely to be present in the clean version of the data. For example, from Figure 3.1, a possible repair would update $t_1[country]$ to "Germany" that would result in the correct pattern: [country = "Germany", capital = "Berlin"] (a pattern is correct if it corresponds to the ground-truth). An alternative repair is to update $t_2[country]$ to "Russia" creating the incorrect pattern [country = "Russia", capital = "Berlin"].

Repairing a violation of an FD may introduce errors that may not even be detectable. For example, in Figure 3.1, updating $t_2[country]$ to "Russia" introduces errors in the data that do not trigger new FD violations. Thus, we need a better way to reason about different repairs beyond the satisfiability of the FDs. In particular, we need to assess the effect of different repairs on the underlying data patterns.

Key Observation. Automatic data-repairing algorithms assume that most of the data is correct. Thus, they strive to change the data minimally to repair violations. This minimality principle has been instilled in various repairing algorithms, e.g., [15–17, 59]. When most of the data is clean, most of the value combinations (e.g., the *FD patterns*) in the data are correct. For instance, in the previous example, the pattern [country = "Germany", capital =

"Berlin"] is strongly supported in the data, making it *likely* correct as opposed to the pattern [country = "Russia", capital = "Berlin"] that is weakly supported in the data.

The Proposed Repair Strategy. In practice, FDs interact with each other through shared attributes. Thus, the FDs' corresponding patterns interact with each other as well. This interaction offers an opportunity to assess the effect of repairing the violation in one FD, say fd_i, on the FD patterns of other FDs that interact with fd_i. For instance, in Example 3.1.1, fd_1 and fd_2 share the attribute *country*. We illustrate how this interaction can be leveraged to reason about the quality of different repairs.

Example 3.1.2 *In Example 3.1.1, we can distinguish two repairs R_1 and R_2 that generate different sets of FD patterns:*

1. *R_1: Update t_2[country] to "Russia". This results in FD patterns p_1 : [cyclist = "Marcel", country = "Russia"] and p_2 :[country = "Russia", capital = "Berlin"] for fd_1 and fd_2, respectively.*

2. *R_2: Update t_1[country] to "Germany". This results in FD patterns p_3 : [cyclist = "Marcel", country = "Germany"] and p_4 : [country = "Germany", capital = "Berlin"] for fd_1 and fd_2, respectively.*

While both R_1 and R_2 result in a consistent instance with respect to fd_1 and fd_2, it is important to dissect the patterns they produce to reason about their quality. In particular, R_1 results in FD patterns p_1 and p_2 that are both supported by one tuple only (t_1) in the original data. R_2 results in FD patterns p_3 and p_4. While p_3 is only supported by one tuple (t_2) in the original data, p_4 is supported by four tuples, making R_2 the *better* repair. Notice that the interaction of fd_1 and fd_2 allows us to consider different value choices for the attribute $country$ in the context of the patterns that carry them. That is, the value of $country$ is part of patterns p_1 and p_2 in R_1 and patterns p_3 and p_4 in R_2. This added context help in identifying the *better* repair R_2.

To help highlight the interplay among FD patterns, we represent each FD pattern by an edge in a dependency graph (refer to Figure 3.1). The graph is the result of instantiating the

FDs on the data. The nodes represent data values and a directed edge from a value v of an attribute X to a value w of an attribute Y exists if there is an FD $X \rightarrow Y$ and the database contains the pair v, w in one of the tuples. The weight of an edge represents its quality that is captured through the number of tuples that support the FD pattern this edge encodes.

The dependency graph in Figure 3.1 illustrates how each choice to repair the violation of fd_1 affects the FD patterns of fd_2 (for clarity of explanation, the graph only includes tuples t_1 to t_5 because the other tuples do not contribute information to fix the violation). Because both choices are supported by one tuple only, looking at the FD patterns of fd_1 in isolation would not provide a good idea about the best value to choose. Looking "beyond" the FD patterns of fd_1 and observing how they affect those of fd_2 would provide a better idea about the value to choose to repair the violation. The correct repair R_2 corresponds to the highlighted path (e_1, e_2). An important implication of this data-repairing approach is that even if we had a majority value to fix fd_1's violation, this majority value may lead to low-quality FD patterns for fd_2. Therefore, looking at the FD patterns collectively is key to producing repairs that preserve data patterns that are strongly supported in the data and hence leading to a better quality repair.

Interpretability. Automatic data repairing needs debugging information for users to make sure the data is clean. While there have been numerous efforts to develop formalisms to express data quality rules [15, 16], little attention has been given to express repairs in a form that facilitates their examination and evaluation. Current repairing algorithms express repairs in terms of the transformations they make to the data [58]. This makes it hard for users to understand and trace the reasons why certain repair decisions were made. An important by-product of our repairing model is the *Interpretability* of its repairs. In particular, when the user wants to trace the decisions that have been made to choose a certain value update, one can easily identify the path in the dependency graph that has led to that repair. This provides the user with a rich context to analyze the chain of patterns that have been involved to produce a certain repair. Furthermore, patterns are more intuitive to analyze than cell values seen in isolation. For instance, in Example 3.1.1, if the user wants to trace the chosen value update R_2, she would be given the path $e_1 \rightarrow e_2$ that produces this value

update. This feature makes it easy to understand which edges have been involved in the decision (e_1 and e_2).

The contributions presented in this chapter are as follows:

- We propose a novel data characterization in the form of FD patterns to model value combinations and their interactions by leveraging *FD* rules. We also define a binary operator to express a dirty data instance and its repairs in terms of its underlying FD patterns.
- We introduce a new class of repairs that aims at maximizing the frequency of FD patterns in the data. We project the FDs over data values to produce a dependency graph, where each edge represents an FD pattern and the edge's weight represents the FD pattern's quality (based on the FD pattern's frequency in the data). We then use this graph to select edges with higher weights to repair tuples.
- We present efficient algorithms to generate repairs in linear time in the size of the data and the FDs. Traversing the dependency graph is driven by a set of heuristics that maximize the quality of the selected edges based on the edges they lead to.
- We express the final instance repair in terms of the FD patterns in the original data. This abstraction makes it easy for users to examine and debug the repair output.
- We provide a thorough experimental study to showcase the performance of our approach compared to a variety of state-of-the-art data repairing algorithms.

3.2 Preliminaries

Let R be a relational schema of a data instance I. Let $A = \{A_1, A_2, ..., A_n\}$ be the set of attributes in R with domains $dom(A_1)$, $dom(A_2)$, ..., $dom(A_n)$ respectively. Let Σ^R be the set of functional dependencies (FDs) defined over R. We say that an instance I of R satisfies Σ^R denoted by $I \models \Sigma^R$ if I has no violations of any of the FDs in Σ^R. We assume that Σ^R is minimal and is in canonical form [60]. In the remainder of the chapter, we refer

to the set of FDs as simply Σ. Let T be the set of tuples in I. $T = \{t_1, t_2, ..., t_n\}$. A cell $t[A]$ denotes the value of attribute A in tuple t. An FD f in Σ^R has the format $X \rightarrow Y$, where $X, Y \in A$. Let $Left(f)$ and $Right(f)$ be the left- and right-hand sides of f, respectively. X and Y are referred to as the antecedent and consequent attributes, respectively. The set of attributes involved in f and Σ are referred to as $attr(f)$ and $attr(\Sigma)$ respectively. When f is projected on a tuple t, we refer to $t[X]$ and $t[Y]$ as LHS and RHS values of f.

Definition 3.2.1 *Repair Instance [17]: Given an instance I of schema R violating FDs Σ^R, an instance I' is a repair of I iff $I' \models \Sigma^R$ and I' retains the same number of tuples as I.*

According to Definition 3.2.1, a repair is achievable only by modifying attribute values of tuples. Insertion or deletion of tuples or attributes are not allowed. Unlike [17], our space of repairs only contains constants from the active domain. There have been numerous efforts to compute repairs that are as close to the clean data as possible [15, 16, 59]. Most existing FD repairing techniques aim at minimizing changes to the data to produce a repair.

Definition 3.2.2 *Cardinality-Minimal Repair [17]. A cardinality-minimal repair I' of Database Instance I differs minimally from I. That is, there is no other repair I'', where $|\Delta(I, I'')| < |\Delta(I, I')|$.*

$\Delta(I, I')$ denotes the set of cells in I that have different values in I'.

Without loss of generality, in our proposal we consider binary distance functions to compute the distance between two data values (1 if two data values are equal and 0 otherwise). Thus, the cost of a repair I', denoted $Cost(I')$, is the number of cells (a specific attribute value in a specific tuple) in the original instance I that are not equal to those in I'.

Definition 3.2.3 *Functional Dependency Graph. A Functional Dependency Graph (FDG) is a directed graph $G(V, E)$, where V contains the set of attribute sets involved in $Left(\Sigma)$ and $Right(\Sigma)$ and E is the set of directed edges, such that $(A_i, A_j) \in E$ iff there is an FD $f(A_i \rightarrow A_j) \in \Sigma$.*

3.3 Modeling Patterns using FDs

In this section, we explain how we project the FDs on the instance to produce value combinations, or FD patterns, of the attributes in the FDs. These patterns constitute the building block of our proposal. We then present the space of repairs we generate.

3.3.1 Functional Dependency Patterns

Data patterns induced by FDs are at the core of our framework. Data is as good as the patterns that constitute it. A wrong value combination at a tuple results in an erroneous tuple. First, we define *simple FD patterns* and discuss their role in our framework. Additionally, simple FD patterns can be composed to embed more than one FD as we show in Section 3.4.3.

Definition 3.3.1 *A simple FD pattern P is a pair (ϕ, V), where (1) ϕ is a single FD from Σ, and (2) V contains a set of pairs (A, a) where $A \in attr(\phi)$ and $a \in dom(A)$. We denote by $P[A]$ the value of pattern P at attribute A, where $A \in attr(\phi)$. To ease the readability of examples, we sometimes omit the name of attributes in V. The antecedent and consequent of P are the attribute values in $Left(\phi)$ and $Right(\phi)$, respectively.*

Though their syntax is similar, FD patterns are fundamentally different from CFDs [56]. The semantics of FD patterns is different from CFDs. FD patterns describe an instance in terms of its FDs and data values, while CFDs are data quality rules meant to be enforced over the instance.

Example 3.3.1 *In Figure 3.1, example FD patterns include:*

- $P_1 : ([fd_1], \{\text{“}Marcel\ Kittel\text{”}, \text{“}Russia\text{”}\})$.
- $P_2 : ([fd_2], \{\text{“}Germany\text{”}, \text{“}Berlin\text{”}\})$.
- $P_3 : ([fd_1], \{\text{“}Paul\ Martens\text{”}, \text{“}Germany\text{”}\})$.
- $P_4 : ([fd_2], \{\text{“}Russia\text{”}, \text{“}Berlin\text{”}\})$.

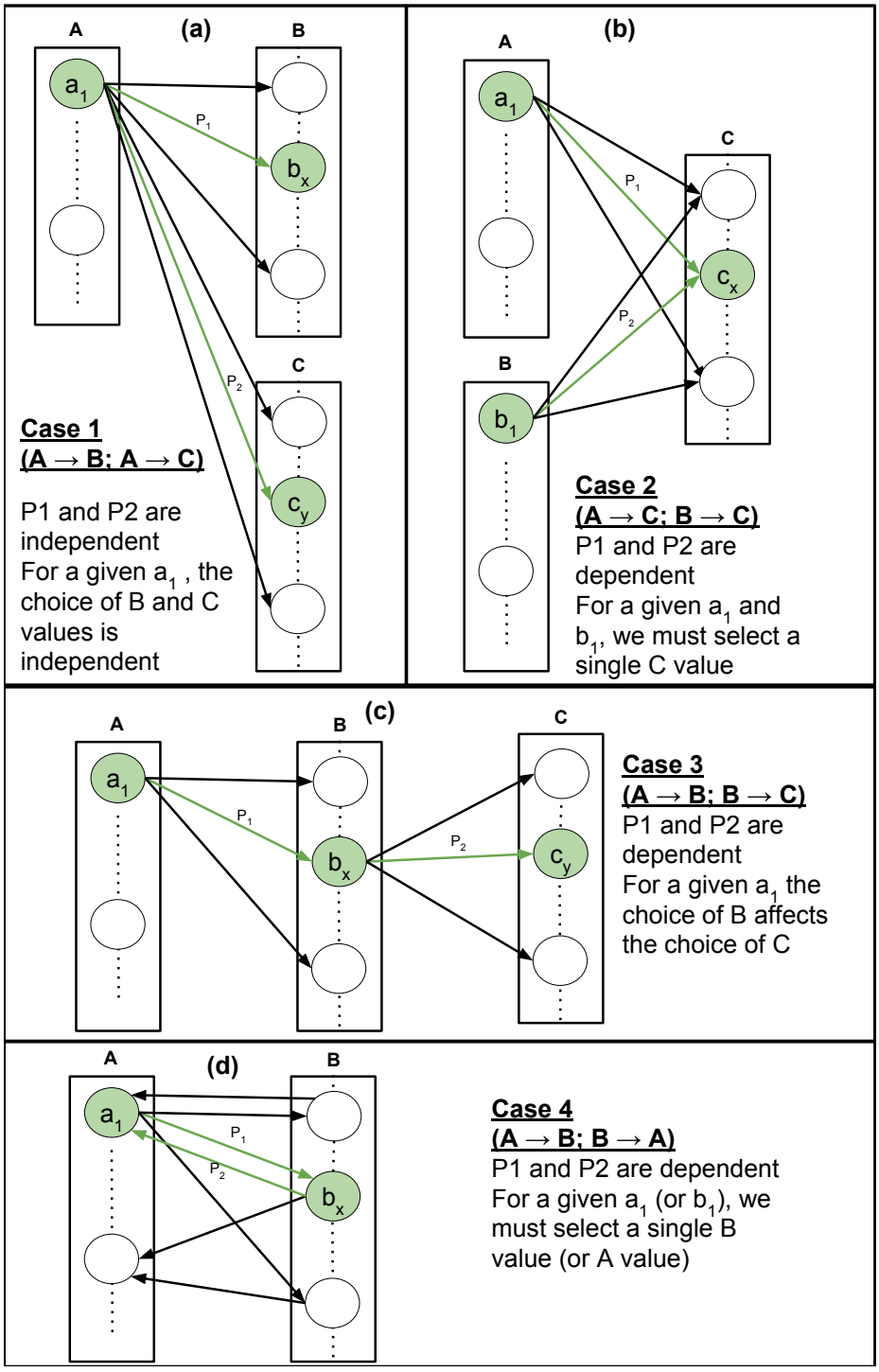


Figure 3.2.: FD Patterns interaction cases

We introduce two metrics to distinguish different instance repairs in terms of their underlying FD patterns.

Instance Quality: The instance quality of I denoted $Q(I)$ containing a set of tuples T is the frequency of each FD pattern in every tuple in T:

$$Q(I) = \sum_{t \in T} \sum_{p \in P(t)} Frequency(p) \tag{3.1}$$

$P(t)$ denotes the set of FD patterns in a given tuple t. *Frequency(p)* is the frequency of pattern $p : (X \rightarrow Y, x, y)$ in I.

Repair Gain: The gain of a repair I' of instance I is the difference in instance quality between I' and I:

$$Gain(I', I) = Q(I') - Q(I) \tag{3.2}$$

The gain of a repair I' is measured by the increase/decrease in frequency of the FD patterns as compared to I. Clearly, we want to compute repairs whose gain is positive.

3.3.2 Problem Definition

As illustrated in Example 3.1.1, it is important to preserve the patterns that are strongly supported by the data. Thus, we extend the cardinality-minimality metric to consider the space of repairs that result in the most supported FD patterns in the data.

Definition 3.3.2 *Pattern-Preserving Repair. A repair I' of Instance I with a cost k is pattern-preserving if there is no repair I'' s.t. (1) $|\Delta(I, I'')| < |\Delta(I, I')|$, and (2) $Gain(I'') > Gain(I')$*

Notice that the first condition can be reduced to the Cardinality-Minimality condition (with k being the minimum cost). Condition (2) ensures that the repair results in the highest repair gain, i.e., the repair has to preserve the FD patterns that are strongly supported in the data.

Proposition 3.3.1 *Computing a pattern-preserving repair is NP-complete for a constant Repair Cost k.*

Proof sketch: The problem of generating pattern-preserving repairs can be reformulated as the the 0/1 knapsack problem for a given cost k. Let I' be the repair that has the maximum repair gain with the repair cost k. That is, there is no other repair I'' such that $Gain(I'', I) > Gain(I', I)$. From Definition 3.3.2, I' also has the maximum instance quality among all other repairs for a given repair cost k. Thus, given the set of all FD Patterns S in I, we want to find a subset $P \subset S$ that maximizes the Repair Quality of a repair I'. In other words, we want to update tuples in I such that the resulting instance I' has the highest Repair Gain for a cost k. Therefore, I' contains, for each tuple, a set of FD patterns corresponding to each FD in Σ such that $Q(I')$ is maximal for a given Repair Cost k.

$$\text{maximize} \sum_{t \in T} \sum_{p \in P} Frequency(p) \text{ subject to: } \sum_{t \in T} \sum_{p \in P} Cost(I') \leq k \tag{3.3}$$

In Section 3.6, we present linear-time repairing algorithms that compute near-optimal pattern-preserving repairs.

Example 3.3.2 *We follow up on Example 3.1.2 to give an example of a pattern-preserving repair. Let I be the $Tour$ instance presented in Figure 3.1. Notice that Repairs R_1 and R_2 are cardinality-minimal with Cost 1. Notice further that there are two other repairs that are cardinality-minimal with Cost 1 besides R_1 and R_2, namely (1) R_3 : change the value of $cyclist$ in t_1 or (2) R_4: change the value of $cyclist$ in t_2. In this example, we only consider R_1 and R_2 for simplicity. For a cost of 1, only R_2 is a pattern-preserving repair. We now show that $Gain(R_2, I) > Gain(R_1, I)$ for $k = 1$.*

Let t_{11} and t_{12} be the tuple t_1 after applying R_1 and R_2 to I, respectively. Similarly, let t_{21} and t_{22} be the tuple t_2 after applying R_1 and R_2 to I, respectively. A summation between brackets denotes the sum of the frequency of each FD pattern at a given tuple.

$Gain(R_2, I) = Q(R_2) - Q(I)$

$Q(R_2) = \underbrace{(2+5)}_{t_{12}} + \underbrace{(2+5)}_{t_{22}} + \underbrace{(1+5)}_{t_3} + \underbrace{(1+5)}_{t_4} + \underbrace{(1+5)}_{t_5} = 32$

$Q(I) = \underbrace{(1+1)}_{t_1} + \underbrace{(1+4)}_{t_2} + \underbrace{(1+4)}_{t_3} + \underbrace{(1+4)}_{t_4} + \underbrace{(1+4)}_{t_5} = 22$

$$\boxed{Gain(R_2, I) = 32 - 22 = 10}$$

$Gain(R_1, I) = Q(R_1) - Q(I)$

$Q(R_1) = \underbrace{(2+2)}_{t_{11}} + \underbrace{(2+2)}_{t_{21}} + \underbrace{(1+3)}_{t_3} + \underbrace{(1+3)}_{t_4} + \underbrace{(1+3)}_{t_5} = 20$

$$\boxed{Gain(R_1, I) = 20 - 22 = -2}$$

Thus, $Gain(R_2, I) > Gain(R_1, I)$.

3.4 FD Pattern Composition and Pattern Expressions

In this section, we study the interactions among FD patterns, and present a formalism to declaratively express repairs in terms of their underlying FD patterns.

3.4.1 Encoding FD Patterns

We encode the FD patterns by projecting the FD graph on the instance. Refer to Figure 3.1 for illustration. Every simple FD pattern $(X \rightarrow Y, [x, y])$ is encoded with a directed edge (x, y). We refer to x and y as the LHS and RHS nodes respectively.

Definition 3.4.1 *The Instance Graph (*IG*) of Instance I is a directed graph, say $G(V, E)$, where: (1) Each node $v \in V$ has two attributes $v.attribute$ and $v.val$ encoding an attribute $a \in A$ and a data value $d \in dom(a)$, respectively; (2) A directed edge $(v, w) \in E$ encodes a simple FD pattern $(X \rightarrow Y, [x, y]) \in I$ such that $v.attribute = X$, $v.val = x$, and $w.attribute = Y$, $w.val = y$.*

For example, the graph illustrated in Figure 3.1 is the instance graph for Instance Tour. In the remainder of the chapter, since the edges in IG encode simple FD patterns, we refer to them as (simple) FD patterns. Additionally, to ease the readability of the graph figures, we label the nodes with their values.

3.4.2 Interactions among FD Patterns

Figure 3.2 enumerates four cases in which FD patterns interact with each other. FD patterns $P_1 : (fd_1, V_1)$ and $P_2 : (fd_2, V_2)$ interact with each other iff: (1) fd_1 and fd_2 share at least one attribute, and (2) the value of the shared attribute(s) between fd_1 and fd_2 is the same in V_1 and V_2. Note that different cases of interactions have different semantics. Consider a dirty tuple t containing two FD patterns P_1 and P_2 corresponding to two different FDs f_1 and fd_2. Without loss of generality, we discuss interaction cases with FDs that have one attribute in their antecedent. P_1 and P_2 can exhibit the following four cases of interaction depending on the FDs they embed (Figure 3.2):

Case 1 ($fd_1 = A \rightarrow B$**,** $fd_2 = A \rightarrow C$**)**: $t[A] = a_1$ can be mapped to any RHS value in B and C, i.e., the choice of values of B is independent of the choice of the value of C. In other words, choosing the RHS of a_1 to satisfy $A \rightarrow B$ does not affect the choice of the RHS of a_1 to satisfy $A \rightarrow C$.

Case 2 ($fd_1 = A \rightarrow C$**,** $fd_2 = B \rightarrow C$**)**: $t[A] = a_1$ and $t[B] = b_1$ must be mapped to the same RHS value C. In other words, Patterns P_1 and P_2 have to share the C value. Thus, the choice of the C value for A affects the choice of the C value for B, and vice-versa.

Case 3 ($fd_1 = A \rightarrow B$**,** $fd_2 = B \rightarrow C$**)**: In this case, the consequent of P_1 is the antecedent of P_2. In this case, the choice of the value of B affects the C value. That is, choosing a value $B = b_x$ in P_1 would make b_x the antecedent of P_2.

Case 4 ($fd_1 = A \rightarrow B$**,** $fd_2 = B \rightarrow A$**)**: This is the case of circular FDs; the choice of the value of A affects the choice of the value of B and vice-versa.

If two patterns P_1 and P_2 interact following any of the above four cases, we say that they are *composable*, denoted by $P_1 \leftrightarrow P_2$. Otherwise, we say they are *not composable*, denoted $P_1 \not\leftrightarrow P_2$.

In the above cases, depending on the interaction case of the FDs, selecting an FD pattern for one FD in a tuple t may affect the choice of the FD patterns for the subsequent FDs that interact with it. We now formalize this observation.

Definition 3.4.2 *Pattern Independence. Two FD patterns $P_1 : (\phi_1, V_1)$ and $P_2 : (\phi_2, V_2)$ are independent if any of the following is true: (1) P_1 and $P2$ exhibit interaction Case 1; (2) The shared attributes between ϕ_1 and ϕ_2 do not carry the same values in V_1 and V_2, respectively; or (3) ϕ_1 and ϕ_2 do not share any attribute. If P_1 and P_2 are not independent, we say they are dependent.*

3.4.3 Composition of FD Patterns

The target is to declaratively describe an instance in terms of its underlying FD patterns. We define the *composition* operator to describe FD patterns whose FDs share one or multiple attributes.

Definition 3.4.3 *Direct Composition Operator. The binary composition operator for FD patterns, denoted by $\triangleright$, is a binary operator such that*

$$P_i \triangleright P_j = \begin{cases} P_{ij} : (\phi_i \cup \phi_j, V_i \cup V_j) \textit{ if } P_i \leftrightarrow P_j \\ \\ P : (\emptyset, \emptyset) \textit{ if } P_i \not\leftrightarrow P_j \end{cases}$$

Intuitively, the binary composition operator allows us to express patterns of multiple FDs when they share some common attributes and the values for these attributes are the same in the composed FD patterns. The binary composition operator is commutative ($P_i \triangleright P_j = P_j \triangleright P_i$) and is left-associative. We refer to FD patterns that embed more than one FD as *composed FD patterns*. The Pattern Independence defined in 3.4.2 for simple FD patterns applies to composed FD patterns as well.

Maximal Composition of FD Patterns: We say that an FD pattern $P : (\phi, V)$ is a maximal composition (or maximal pattern) w.r.t. a set of FDs in Σ if there is no simple FD pattern $p_i : (f_i, v_i)$, where P_i is composable with P and $f_i \notin \phi$. In other words, P should contain a composition of all the simple FD patterns that can interact with each other (based on Interaction Cases 1-4).

Example 3.4.1 *Consider the FD patterns in Example 3.3.1.* P_1 *is composable with* P_4 *(*$P_1 \leftrightarrow P_4$*) and their composition produces a composed FD pattern* $P_{14} = P_1 \rhd P_4 = ([fd_1, fd_2], cyclist$*="Marcel Kittel",* $country =$ "$Russia$", $capital =$ "$Berlin$"*).* P_{14} *is also a maximal composition w.r.t.* fd_1 *and* fd_2*.*

Notice that P_1 *is not composable with* P_2 *(*$P_1 \not\leftrightarrow P_2$*) because they have different values in the common attribute* $capital$*.*

3.4.4 Pattern Expressions

In this section, we show how we can describe any instance as a composition of its underlying simple FD patterns.

Definition 3.4.4 *A pattern expression for a tuple* t*, denoted by* $P^{exp}(t)$ *contains the set* S *of maximal FD patterns such that* S *covers all FDs in* Σ*.*

Because all FD patterns in a pattern expression are maximal, it follows that a pattern expression for a tuple contains independent FD patterns. Pattern expressions are particularly useful to express the repair instance. The reason is that they enable users to see a repaired tuple in terms of the FD patterns in the original data that have been composed to produce the tuple. This facilitates the interpretability of the repairs because users can trace repair decisions in terms of the edges (or FD patterns) in the Instance Graph.

Example 3.4.2 *We build on Example 3.1.2 to generate repair expressions for the instance* $Tour$ *in Figure 3.1. We complement the set of FD patterns in Example 3.1.2 to include those of tuples* t_3*,* t_4 *and* t_5 *as follows:* P_5 *: [cyclist = "Andre Greipel", country = "Germany"],* P_6 *: [cyclist = "Emmanuel Buchmann", country = "Germany"],* P_7 *: [cyclist = "Paul Martens", country = "Germany"]. The repair expressions for the tuples in Table* $Tour$ *are as follows:*

$P^{exp}(t_1) = \{P_3 \rhd P_4\}$; $P^{exp}(t_2) = \{P_3 \rhd P_4\}$; $P^{exp}(t_3) = \{P_5 \rhd P_4\}$; $P^{exp}(t_4) = \{P_6 \rhd P_4\}$; $P^{exp}(t_5) = \{P_7 \rhd P_4\}$

Table 3.1.: Extended instance *Tour_rank*

	rank	**cyclist**	**country**	**capital**
t_1	166	Marcel Kittel	Russia	Berlin
t_2	166	Marcel Kittel	Germany	Berlin
t_3	166	Andre Greipel	Germany	Berlin
t_4	133	Andre Greipel	Germany	Berlin
t_5	21	Emanuel Buchmann	Germany	Berlin
t_6	98	Paul Martens	Germany	Berlin

3.5 Pattern Quality

Our target is to select "good" FD patterns in the instance graph to compute instance repairs. Therefore, it is crucial to characterize the quality of simple FD patterns in the instance graph. This step is required by the repair algorithm (Section 3.6) to reason about the quality of the various candidate FD patterns. We presented in Section 3.3 simple quality metrics of FD patterns based on their frequency in the data. We now present a general model to characterize the quality of FD patterns that also captures their interaction. Based on well-known frequency-based metrics defined for association rules [61], we present several metrics to capture the quality of FD patterns (and the ones they affect) in the instance graph. By looking at a simple FD pattern $P : (X \rightarrow Y, [x, y])$ as an association rule $(P[x] \rightarrow P[y])$, its $Support$ is the number of tuples with $X = x$ and $Y = y$ in I over the number of tuples in I. The $Confidence$ of P is the number of tuples with $X = x$ and $Y = y$ in I over the number of tuples with $X = x$ in I [61].

$$Conf(P) = \frac{|P|}{|(X \rightarrow Y, [x, *])|} \tag{3.4}$$

$$Sup(P) = \frac{|P|}{|(X \rightarrow Y, *, *)|} \tag{3.5}$$

* denotes "any value". $|(X \rightarrow Y, [x, *])|$ denotes the number of tuples in I with the LHS value x and any RHS value.

As illustrated in Example 3.1.1, greedily selecting FD patterns based on their frequencies is not a good strategy for selecting the best FD patterns. It is better if the score of an

FD pattern not only includes its own confidence and support, but also the confidence and support of the FD patterns it can lead to. Thus, we extend Equations 3.4 and 3.5 to capture the quality of the FD patterns that can be reached from a simple FD pattern P. We define the quality of a simple FD pattern P by the set of FD patterns it can lead to (denoted $P^{\rightarrow}$) as follows:

$$Score(P) = Conf(P) + Sup(P) + \sum_{Q \in P^{\rightarrow}} Conf(Q) + \sum_{Q \in P^{\rightarrow}} Sup(Q) \tag{3.6}$$

$$Quality(P) = \frac{Score(P)}{2(|P^{\rightarrow}| + 1)} \tag{3.7}$$

$Score(P)$ (Equation 3.6)) is the sum of: (1) the $Support$ and $Confidence$ of P, and (2) the $Support$ and $Confidence$ of all the simple FD patterns that can be reached from P. We normalize the score of a pattern using the average over the number of edges in $|P^{\rightarrow}|$ (we multiply it by two since every edge embeds Sup and $Conf$) (Equation 3.7). One can normalize $Score(P)$ using other aggregate functions, but we found that the average captures well the quality of simple FD patterns.

So, far, we have presented quality metrics for a simple FD pattern that corresponds to an edge in the instance graph. We generalize Equation 3.7 to define the quality of a composed FD pattern Q as follows ($|Q|$ denotes the number of simple FD patterns in Q):

$$Quality(Q) = \frac{\sum_{p \in Q} Quality(p)}{|Q|} \tag{3.8}$$

Algorithm 4 shows the pseudocode to compute the quality of simple FD patterns in IG. It uses Algorithm 4 to traverse IG. Algorithm 3 performs a Depth-First Search (DFS) traversal over IG, and computes the quality of each visited edge and vertex. The quality of an edge is computed according to Equation 3.7. The quality of a vertex v is the average quality of all the edges that can be reached from v. To guarantee termination, back-edges (those that correspond to cyclic FDs) are processed when the DFS traversal is complete. Specifically, Algorithm 3 performs the following steps: (1) Build a DFS tree from the input root vertex v; (2) For every edge $e = (v, w)$, if e is a back-edge, it is added to a set

Algorithm 3: Traverse(Vertex v)

```
   output: Average quality of edges starting from Input Vertex v
1  if v.adjacent() = ∅ then
2      vQuality[v] ←0
3  forall w ∈ v.adjacent() do
4      Edge e = (v, w)
5      Score ← 0
6      if visited[w] = true then
7          BackEdges ← BackEdges ∪ e
8          return 0
9      else
10         visited[w] ← true
11         Score ← Conf(e) + Sup(e) + Traverse(w)
12         Quality[e] ← Score / (2(|e→|+1))
13         vQuality[v] ← vQuality[v]+ Score
14 return vQuality[v]
```

$BackEdges$ (Line 7). If not, compute the edge quality (Line 12); (3) compute the quality of the root vertex v (Line 13). After the DFS step is completed, all back-edges are processed (Algorithm 4, Lines 7-8). The quality of a back-edge $e = (v, w)$ is the average of three values: the quality of vertex w (computed in the DFS step), $Conf(e)$ and $Sup(e)$.

Complexity: Given an instance graph $IG(V, E)$, the time and space complexities of Algorithm 3 are both $O(|V| + |E|)$.

Example 3.5.1 *Consider the instance Tour_rank in Table 3.1. Consider the FDs defined in Example 3.1.1 and add the following ones:* $fd_3 : rank \rightarrow cyclist$; $fd_4 : cyclist \rightarrow rank$. *Figure 3.3 illustrates the Instance Graph obtained from Instance Tour_rank with the edge quality computed using Algorithm 4. For instance, Quality*(e_1) = *avg[Sup*(e_1) + *Conf*(e_1) + *Quality*(v_2)] = *avg[Sup*(e_1) + *Conf*(e_1) + (*Sup*(e_3) + *Conf*(e_3)) + (*Sup*(e_4) + *Conf*(e_4)) + *Quality*(v_3) *Quality*(v_4)] = *avg[Sup*(e_1) + *Conf*(e_1) + (*Sup*(e_3) + *Conf*(e_3)) + (*Sup*(e_4) + *Conf*(e_4)) + (*Sup*(e_5) + *Conf*(e_5)) + (*Sup*(e_6) + *Conf*(e_6))] = $[(0.4 + 0.66) + (0.2 + 0.5) + (0.2 + 0.5) + (0.2 + 1) + (0.8 + 1)]/10 = 0.54$

Algorithm 4: ComputePatternsQuality(FD Pattern Graph G)

output: Edge-weights reflecting the quality of the FD patterns

```
1 BackEdges ← ∅
2 V ← G.vertices
3 for i ← 0 to |V| do
4 |   visited[i] ← false
5 forall v ∈ V do
6 |   Traverse(v)
7 forall Edge e(v, w) ∈ BackEdges do
8 |   Quality[e] ← avg(Sup(e), Conf(e), vQuality[w])
```

Another example is to compute the quality of the back-edge e_2*:* $Quality(e_2)$ = *avg[*$Sup(e_2)$ + $Conf(e_2)$ + $Quality(v_2)$*]* = *avg[*$Sup(e_2)$ + $Conf(e_2)$ + *(Sup(*e_3*)* + *Conf(*e_3*))* + *(Sup(*e_4*)* + *Conf(*e_4*))* + *(Sup(*e_5*)* + *Conf(*e_5*))* + *(Sup(*e_6*)* + *Conf(*e_6*))]* = $[(0.4 + 0.1) + (0.2 + 0.5) + (0.2 + 0.5) + (0.2 + 1) + (0.8 + 1)]/10 = 0.49$

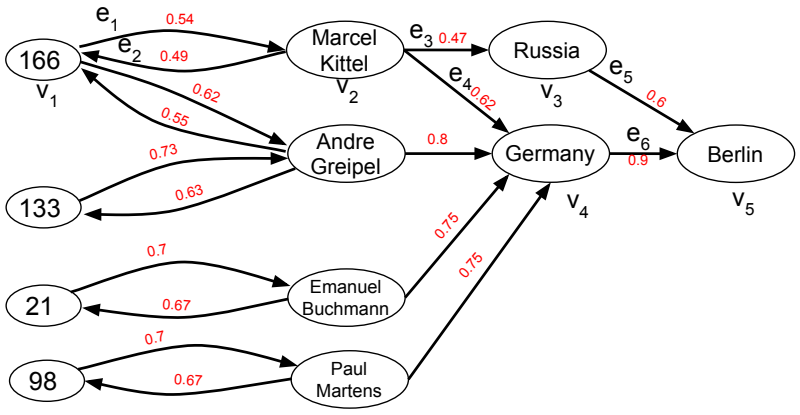


Figure 3.3.: Instance graph of instance Tour_rank with quality scores

3.6 Traversing the Instance Graph for Data Repairing

An important step in data repairing is to decide which cell values in the input tuples should be retained and which ones should be modified. We classify the attributes involved

in the FDs as *bounded* (attributes whose values cannot change) or *free* (attributes whose values can be changed). This is a reasonable assumption made in prior repairing algorithms, e.g., [57] and [18] to limit the scope of changes by the repair algorithm.

3.6.1 Determining Bounded and Free Attributes

FDs impose a "many-to-one" relationship between LHS and RHS values. That is, for the instance to be consistent, a LHS value is mapped with a single RHS value. An attribute A that does not appear as a RHS of an FD is said to be a *bounded* attribute. Bounded attributes have two properties: (1) They appear as part of the LHS in Σ and are thus used to *determine* the value of RHS attributes, and (2) Since they do not appear as RHS attributes in Σ, we cannot use other attributes to determine their values (because of the many-to-one relationship, we can only determine attribute values from LHS to RHS and not the other way around). If an attribute is not *bounded*, then, it is a *free* attribute, i.e., its values are determined from other attributes. Obviously, an attribute cannot be $bounded$ and $free$ at the same time. Therefore, all *free* attributes must appear as RHS attributes in Σ (we discuss the case of cyclic FDs next).

Proposition 3.6.1 *For every $free$ attribute A in Σ, there must exist at least an attribute B such that: (1) There is an FD $\phi_1(B \rightarrow A) \in \Sigma$ (2) If there is an FD $\phi_2(A \rightarrow B) \in \Sigma$, then, there must exist at least an FD $\phi_3 \in \Sigma$ where $\phi_3(C \rightarrow A)$ or $\phi_3(C \rightarrow B)$. If $\phi_3 \notin \Sigma$, then we designate either A or B to be a $bounded$ attribute.*

Proposition 3.6.1 states that every RHS attribute (free attribute) has to have at least one set of LHS attributes that determines it in Σ. This proposition is trivial when there are no cyclic FDs in Σ. However, if Σ contains cyclic FDs, some attributes could be $free$ but would not have an LHS attribute that determines them outside the cycle. For instance, consider Example 3.5.1. $rank$ and $cyclist$ are both *free* attributes (they appear as RHS attributes), but they do not have LHS attribute outside the cycle that determines either one of them (Condition 2 in Proposition 3.6.1). In this case, we randomly pick one of the attributes involved in the cycle to be a $bounded$ attribute (and not a $free$ attribute) and use

it to determine the remaining nodes in the cycle. As a result, the value of this attribute is taken from the input tuples. For example, one could choose attributes $cyclist$ or $rank$ in Example 3.5.1 to be $bounded$ attributes. This way, we can set the value of one attribute to determine the value of other attributes.

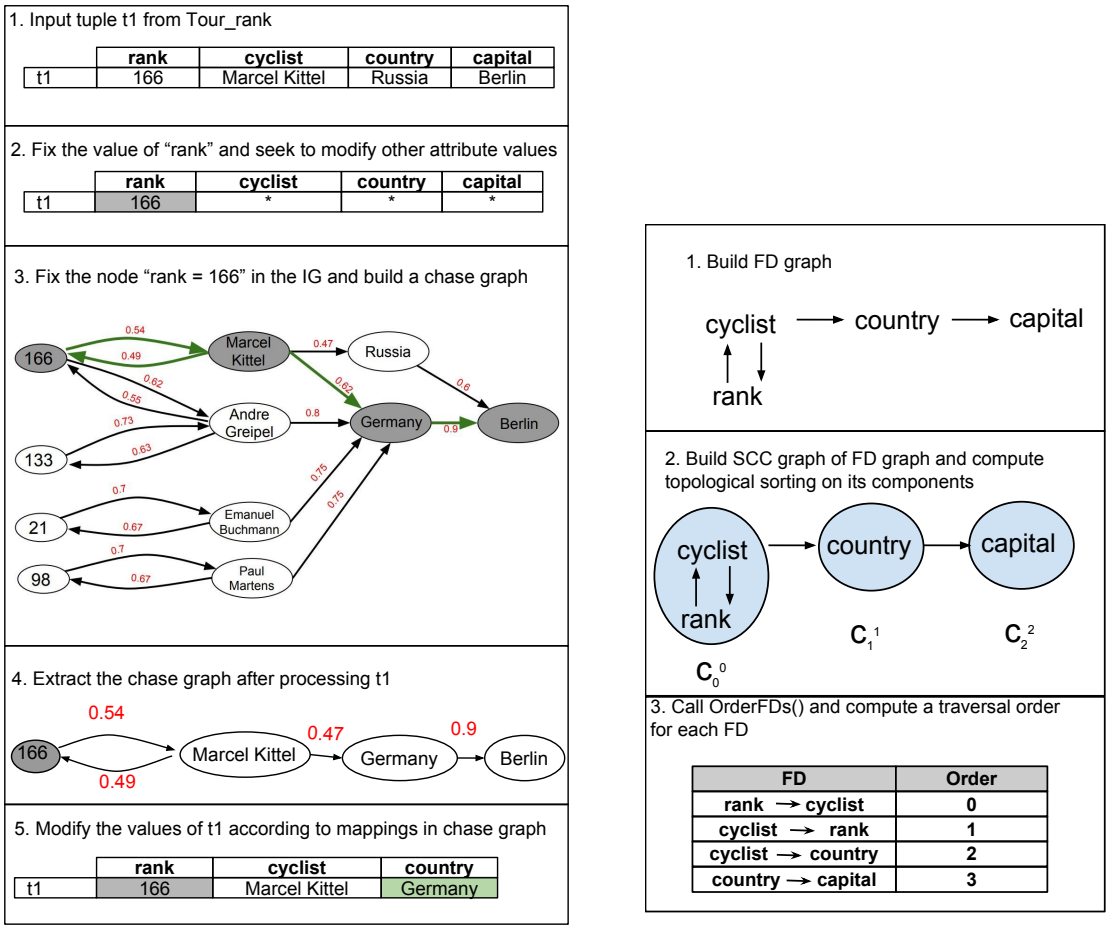
Example 3.6.1 *Consider the following FDs: $A, B \rightarrow C$; $C, D \rightarrow E$. The $free$ attributes are C, E and the $bounded$ attributes are A, B, D. C and E are $free$ because they appear as RHS attributes in the FDs. A, B, and D are $bounded$ attributes because they do not appear as RHS attributes in any FD.*

Consider another set of FDs, where we have cycles: $A \rightarrow B$; $B \rightarrow A$. Both A and B appear as RHS attributes. In this case, we have to choose one attribute from the ones involved in the FD cycle (i.e., A or B) to be a $bounded$ attribute and the other would be $free$.

Consider the following FDs: $E \rightarrow A$; $A \rightarrow B$; $B \rightarrow A$. E is a $bounded$ attribute and both A and B are $free$ attributes. Notice that in this case, even though A and B are involved in a cycle, we do not have to make either one of them a $bounded$ attribute because there is an FD ($E \rightarrow A$) (Condition 2 in Proposition 3.6.1).

3.6.2 Instance Graph Traversal using Attribute Boundedness

As stated in Definition 3.4.1, every value v from Attribute A is represented as a node, say n, where $n.val = v$ and $n.attribute = A$. Consequently, the node values coming from bounded attributes are assigned from the input tuples. For example, consider instance *Tour_rank* in Table 3.1 and its corresponding IG in Figure 3.3. The set of $bounded$ attributes can contain either $cyclist$ or $rank$. Assume that we choose $cyclist$ to be the $bounded$ attribute. The rest of the attributes are $free$. Given Tuple t_1 in Instance *Tour_rank*, one would "fix" the value $t_1[cyclist]$ that corresponds to the node labeled "Marcel Kittel" in IG in Figure 3.3. Starting from this node, we follow (or chase) the edges in IG for each FD in Σ (details about the graph traversal are discussed in the next section). This traversal



(a) Repairing a tuple using the chase graph

(b) Example of computing the FD order

Figure 3.4.: Example of the steps taken to repair a tuple

produces a single FD pattern for each FD in Σ. We call the subgraph induced by this traversal the *Chase Graph*.

Proposition 3.6.2 *For a given assignment β of bounded attribute nodes A in the $IG(V, E)$, there exists a subgraph $G(T, Y)$ such that: (1) $T \subset V$ and $Y \subset E$ and $A \subset T$; (2) $\forall \phi(X \rightarrow Y) \in \Sigma : \exists e(V, W) \in E : V.attribute = X \wedge W.attribute = Y$.*

Proposition 3.6.2 states that assigning values to the $bounded$ attribute nodes in the IG produces a subgraph (the chase graph) that covers all the FDs in Σ. In other words, the set of bounded attribute values is all we need to determine the value of all the other attributes in Σ. Figure 3.4 illustrates the chase graph generated with $rank$ as a bounded attribute. For

example, given Tuple $t1$ (a), the assignment of the bounded attribute is $\beta = \{t1[rank] = 166\}$, all the other attributes can be modified (b). Then, we start the chase to get the FD patterns of the other FDs from IG (c). Then, the resulting chase graph (d) is used to repair Tuple $t1$ (e). We discuss strategies and details for the traversal in the next section.

3.6.3 Repair Covers

Based on the classification of FD attributes (i.e., bounded or free), we observe a few properties that can be leveraged to repair the data. When we have FD patterns that interact with each other, the choice of value for one attribute affects the FD patterns in which that attribute appears. In other words, every node in the FD graph influences the FD patterns it belongs to. Refer to Example 3.1.1 for illustration. Attribute "country" is involved in two FDs, and hence it influences the FD patterns of both FDs. For instance, in Figure 3.1, the value [country="Russia"] affects FD patterns ([cyclist = "Marcel Kittel", country = "Russia"] and [country = "Russia", capital = "Berlin"]). Thus, when choosing a value of "country" to associate with the LHS attribute "cyclist", one should consider FD patterns from fd_1 and fd_2. Ideally, we want to choose FD patterns with the maximum quality.

Therefore, given the FD graph and the set of $bounded$ and $free$ attributes, we can determine for each FD attribute the set of FD patterns it influences. Consider Example 3.1.1. Attribute "country" influences the FD patterns of fd_1 and fd_2.

Definition 3.6.1 *Given an FD Graph $G(V, E)$ and a Node $n \in V$, the Repair Cover $RC(n)$ is defined as: (1) If n.attribute is free, then $RC(n)$ contains for each FD in Σ a single edge e:(x, y) in IG, where $x = n$ or $y = n$, (2) If n.attribute is bounded, then $RC(n)$ is empty.*

Intuitively, a repair cover of a node n contains all the edges (or simple FD patterns) in IG, where n is involved (one for each FD) if $n.attribute$ is $free$. Since we cannot change data values of $bounded$ attributes, if $n.attribute$ is $bounded$, then its repair cover is empty. Note that the repair cover of a node is not unique, and a node typically has multiple repair

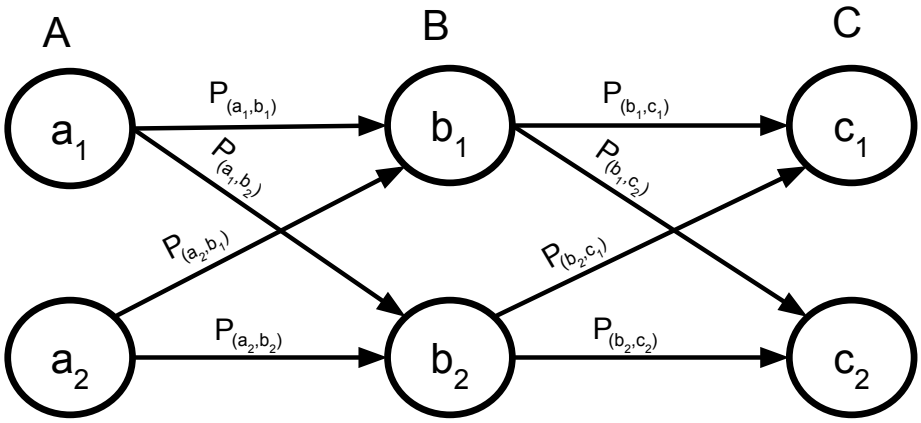


Figure 3.5.: Example Instance Graph for FDs A $\rightarrow$ B and B $\rightarrow$ C

covers involving different simple FD patterns. Additionally, repair covers can be expressed as a composition of simple FD patterns, and their quality is computed as in equation 3.8.

The purpose of a repair cover is to assess the quality of the "neighborhood" of a candidate RHS node before mapping it to an LHS node. Thus, the repair cover contains all the edges n is involved in. As a result, if $n.value$ is a "correct" value, then, it should be connected to high-quality edges, if not, then n should lower the quality of its adjacent edges.

Example 3.6.2 *Figure 3.5 gives an example instance graph for the FDs: $A \rightarrow B$ and $B \rightarrow C$. The repair covers for some nodes in the example instance graph (we put different possible repair covers in a set):*

- $RC(a_1) = \emptyset$, $RC(a_2) = \emptyset$
- $RC(b_1) = \{P_{(a_1,b_1)} \rhd P_{(b_1,c_1)}, P_{(a_1,b_1)} \rhd P_{(b_1,c_2)}, P_{(a_1,b_1)} \rhd P_{(b_1,c_2)}, P_{(a_2,b_1)} \rhd P_{(b_1,c_1)}, P_{(a_2,b_1)} \rhd P_{(b_1,c_2)}\}$
- $RC(c_2) = \{P_{(b_1,c_2)}, P_{(b_2,c_2)}\}$

3.6.4 Optimal Pattern-Preserving Data Repairing

Computing the optimal pattern-preserving repair requires finding a chase graph with maximum edge weights for each assignment of bounded attributes. Let U be the set of bounded attributes, and D be the free ones. Also, let d be the number of distinct values of each attribute $A \in attr(\Sigma)$ (we assume we have d distinct values in each attribute A). Finding the chase graph in IG with the highest sum of weights requires traversing all the graph nodes for each assignment of bounded nodes (Proposition 3.6.2). We have $d^{|U|}$ assignments, where each is chased in IG to associate a single RHS node for every LHS node. Thus, the complexity is $O(d^{|U|} * d^{|D|}) = O(d^{|A|})$. Next, we present heuristics to find a near-optimal Pattern-Preserving repair in linear time.

3.6.5 Traversing the Instance Graph

We present the necessary building blocks to traverse the Instance Graph and compute repairs in linear time. In order to clean to the data, we need to map every LHS node in IG to a single RHS node. This imposes a traversal order going from LHS nodes to RHS nodes. Ideally, we want to start traversing IG from the leftmost attributes in Σ, and chase the adjacent nodes until we build a chase graph for each input tuple. To devise a traversal order to generate the chase graphs, we start chasing the nodes from the leftmost attributes in Σ to the rightmost ones. This linear ordering can be obtained by applying topological sort on the FD graph. However, since cyclic FDs in Σ is possible, i.e., FDs whose RHS attribute appears in the LHS of another FD, we cannot apply topological sort directly. Instead, we apply topological sort on the Strongly Connected Component Graph (SCCG) induced by the FD graph. We obtain the SCCG using $Tarjan's$ algorithm [62] that runs in $O(|A| + |E|)$, where A and E are the vertices and edges in the FDG. Let $SCCG(C, E)$ be the SCC graph induced by the FD graph, where C is the set of SCCs in the FD graph, and E is the set of edges connecting them. Applying topological sort on

SCCG produces a partial order Γ on the SCCs in C. Formally, a SCC $c_i \in C$ is assigned an order o, denoted by c_i^o as follows:

$$\Gamma = \begin{cases} c_i^0 = \{c \in C | \forall c' \in C : (c', c) \notin E\} \\ \\ c_i^{i+1} = \{c \in C | \forall c' \in C, (c', c) \in E : c' \in C_j\} \text{ s.t. } j \leq i \end{cases}$$

Note that multiple SCCs can have the same topological order.

Our traversal of IG is driven by the set of FDs in Σ. More specifically, for a given input tuple t, we choose the "best" FD pattern (from IG) for every FD in Σ, and then insert these patterns in t. Hence, we want to assign orders to all the FDs in Σ, which correspond to the edges in the FD graph. Thus, we introduce a function *OrderFDs* in Algorithm 5 that takes as input the ordered SCCs (OC) computed using Γ and assigns an order to each FD in Σ.

Algorithm 5: OrderFDs(Σ, OC)

```
   output: Array A of FDs, where A[k] contains FDs with order k
1  A ← []
2  k ← 0
3  V ← G.vertices
4  for (i ← 0; i ¡ |OC|; i++) do
5      c ← OC[i]
6      forall e(v, w) ∈ IN(c) do
7          ω ← GetFD(Σ, e)
8          A[k] ← ω
9          k ← k + 1
10     forall e(v, w) ∈ OUT(c) do
11         ω ← GetFD(Σ, e)
12         A[k] ← ω
13     k ← k + 1
```

Algorithm 5 computes the order of every FD in Σ. The output is an array A where $A[k]$ contains FDs with order k. The algorithm proceeds as follows: Since we have to visit all the edges inside an SCC c ($IN(c)$) before visiting c's adjacent SCCs in the SCCG, we incrementally assign an order to each edge (that corresponds to an FD) inside c (Lines

6-9). Note that it does not matter which FD to visit first inside c (all nodes can be reached from any node in c). Next, the algorithm assigns an order k to the outgoing edges from c ($OUT(c)$). Since outgoing edges should only be traversed after traversing all the edges in $IN(c)$, k has to be greater than the highest order assigned to an edge in $IN(C)$. Additionally, edges in $OUT(c)$ can be visited after traversing all the edges in $IN(c)$. Therefore, edges in $OUT(c)$ share the same order (Lines 10-12). Figure 3.4(b) illustrates how Algorithm 5 orders the FDs in Example 3.5.1. In particular, we perform the following steps: (1) From Σ, compute the FD Graph; (2) Compute the SCCG and its topological sorting using Γ, and (3) Compute the order of the FDs in Σ using Algorithm 5.

3.6.6 Pattern-Preserving Repair Algorithms

We present a repair algorithm that computes a repair instance in the form of pattern expressions. Notice that a pattern expression corresponds to a *Chase Graph* in IG. Our final goal is to choose Chase Graphs that have heavy weights on their edges without resorting to an exponential solution.

Algorithm 6 takes as input a dirty table D and the set of FDs Σ, and produces as output pattern expressions that correspond to clean tuples. Mappings of LHS to RHS values are stored in tables (termed Repair Tables). Every FD has its own Repair Table that contains (LHS, RHS) mappings produced by the repair algorithm. Since the algorithm proceeds one tuple at a time, these tables are required to check if an LHS value has been assigned an RHS value in a previous iteration. Repair tables are used to update the input tuples accordingly. This part is straightforward, and is omitted for brevity. First, we build the Instance Graph and compute its edge weights (Lines 1-2), and compute the partial order of FDs (Lines 3-5). The algorithm processes the input data tuple at a time (Line 7), and creates a pattern expression $P^{exp}(t)$ for each Tuple t by building the chase graph from IG (Lines 18-21). Then, the (LHS, RHS) mappings are written into the repair tables of each corresponding FD (Lines 17 and 21). We traverse the set of ordered FDs (Line 9), and assign a RHS value to the LHS value found in the input tuple. If this LHS is already mapped to a RHS value

Algorithm 6: GeneratePatternPreservingRepairs(Σ, D)

output: For every tuple in D, return a pattern expression

IG ← BuildInstanceGraph(D, Σ)
IG ← ComputePatternsQuality(IG)
SCCG ← BuildSCCGraph(Σ)
OC ← TopologicalSorting(SCCG)
Ordered_FDs ← OrderFDs(Σ, OC)
pattern_expressions ← $\emptyset$
forall *Tuple t* $\in D$ **do**
    for $i \leftarrow 0$ to $|Ordered\_FDs|$ **do**
        forall *FD f* $\in Ordered\_FDs[i]$ **do**
            Lval ← t[f.LHS]
            if *Rtable(f).contains(Lval)* **then**
                FDPattern p ← New FDPattern(f, Lval → Rtable(f).get(Lval))
                $P^{exp}(t) \leftarrow P^{exp}(t) \rhd$ p
            else if *f.RHS* $\in P^{exp}(t)$ **then**
                FDPattern p ← New FDPattern(f, Lval →
                  GetAttributeValue($P^{exp}(t)$, f.RHS))
                $P^{exp}(t) \leftarrow P^{exp}(t) \rhd$ p
                Rtable(f).Add(Lval, GetAttributeValue($P^{exp}(t)$, f.RHS))
            else
                FDPattern p ← Edge_Selection(IG)
                $P^{exp}(t) \leftarrow P^{exp}(t) \rhd$ p
                Rtable(f).Add(Lval, p.RHS)
    pattern_expressions = pattern_expressions $\cup P^{exp}(t)$

(Line 11), then we fetch this mapping from the repair table, build a pattern (Line 12), and then add this pattern to the pattern expression using the composition operator (Line 13). If the RHS attribute of the current FD has been assigned a value in the pattern expression P^{exp} (Line 14), then, we cannot replace it with another value (pattern interaction Case 2). In this case, we map the current LHS to that RHS value (Line 15) and the pattern is added to the pattern expression (Line 16). The last case (Line 18) is when the LHS value has not been assigned a RHS value. In this case, we fetch the "best" RHS value from the IG (Line 19). Depending on the edge-selection strategy (presented next), we may get different patterns. Then, the pattern is added to the pattern expression (Line 22).

Edge Selection Strategies: We implement three strategies to map, for a given FD $(X \rightarrow Y)$, a LHS node to a RHS node in IG: (1) *Greedy*: This heuristic performs a greedy traversal of the Instance Graph. Given an LHS node, we choose the adjacent edge with the highest quality. Notice that this is not a trivial traversal of IG, as it still benefits from the quality scores and the ordered traversal of IG. In fact, experiment results show that this heuristic performs in some cases better than *RC*. (2) *RC-based Traversal (RC)*: The repair cover of all the adjacent RHS nodes is computed. Then, the RC with the highest score is selected to map an LHS node to an RHS node. This traversal evaluates the neighborhood of the adjacent nodes (and the nodes they lead to), before selecting an RHS node. (3) *Hybrid*: A hybrid of the previous two, this heuristic decides, given an LHS node and Threshold θ, whether to choose an RHS node by only looking at the adjacent edges (Greedy) or compute the repair covers of the RHS nodes, and then decide which RHS nodes to select (RC). The intuition is that if the adjacent edges have a high-enough score (i.e., $\geq \theta$), then looking at their neighbors could decrease the quality of the repair. This is especially relevant when we have an idea about the error rate in the data. Experiments demonstrate that this heuristic outperforms *Greedy* and *RC* in repair quality.

Discussion on Repair Requirements

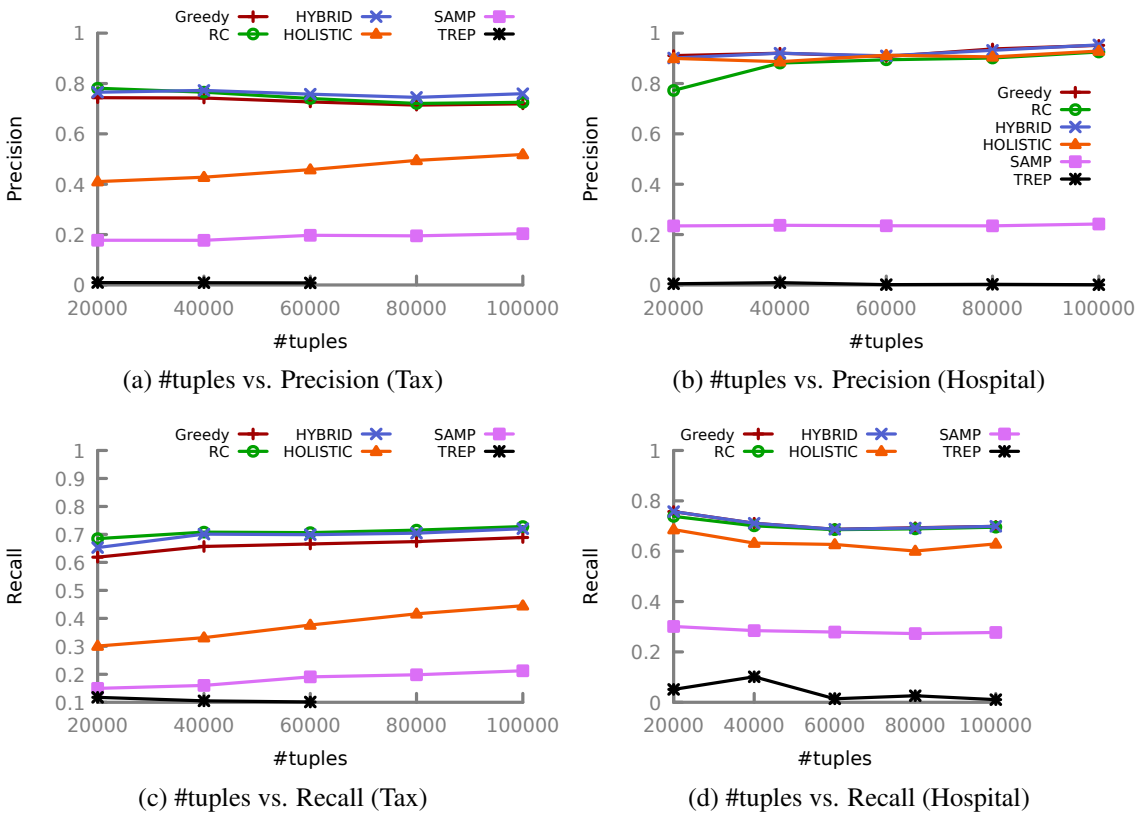
We highlight key properties of Algorithm 6. **FD Requirement:** The algorithm ensures that every LHS value is mapped to a single RHS through the repair tables. Particularly, if the LHS has been previously mapped to an RHS value, that mapping is used in the pattern expression (Lines 11-13). In case we have a backedge in the FD graph, i.e., an attribute appears as both an LHS and RHS in Σ, the algorithm (Lines 14-17) makes sure that an LHS value is only mapped to an RHS value in the current pattern expression. **Soundness:** The weight computation of FD patterns (Line 2) offers a key parameter for the edge-selection strategies. We have three examples of edge-selection strategies, mainly, greedy, RC-based, and Hybrid. **Coverage:** The algorithm computes a pattern expression for every input tuple t. **Termination:** For every tuple, the traversal of IG is bounded by the number of FDs in Σ (Line 8), due to the topological sort of the FD graph (Line 3-4) that imposes an order on the traversal of IG (Line 5). **Interpretability:** The output of Algorithm 6 is a pattern expression for each tuple t. Users can use this output to trace repairs to their underlying FD patterns.

3.7 Experimental Study

We evaluate our repair approach against other repairing algorithms. In this section, we present and discuss the experimental results.

3.7.1 Setup

Dataset. We use the following two datasets:(1) A synthetic dataset [56] that contains records pertaining to tax information for different persons, e.g., *first name, last name*, and *whether the person has a child*. For measuring effectiveness, we use 100K tuples (*Tax*) while we generate millions of tuples for the scalability study (*Tax_Extended*). (2) *Hospital* is a real-world dataset that contains information about health-care providers and hospitals. It contains 100K tuples. We define four FDs for each dataset.



(a) #tuples vs. Precision (Tax)

(b) #tuples vs. Precision (Hospital)

(c) #tuples vs. Recall (Tax)

(d) #tuples vs. Recall (Hospital)

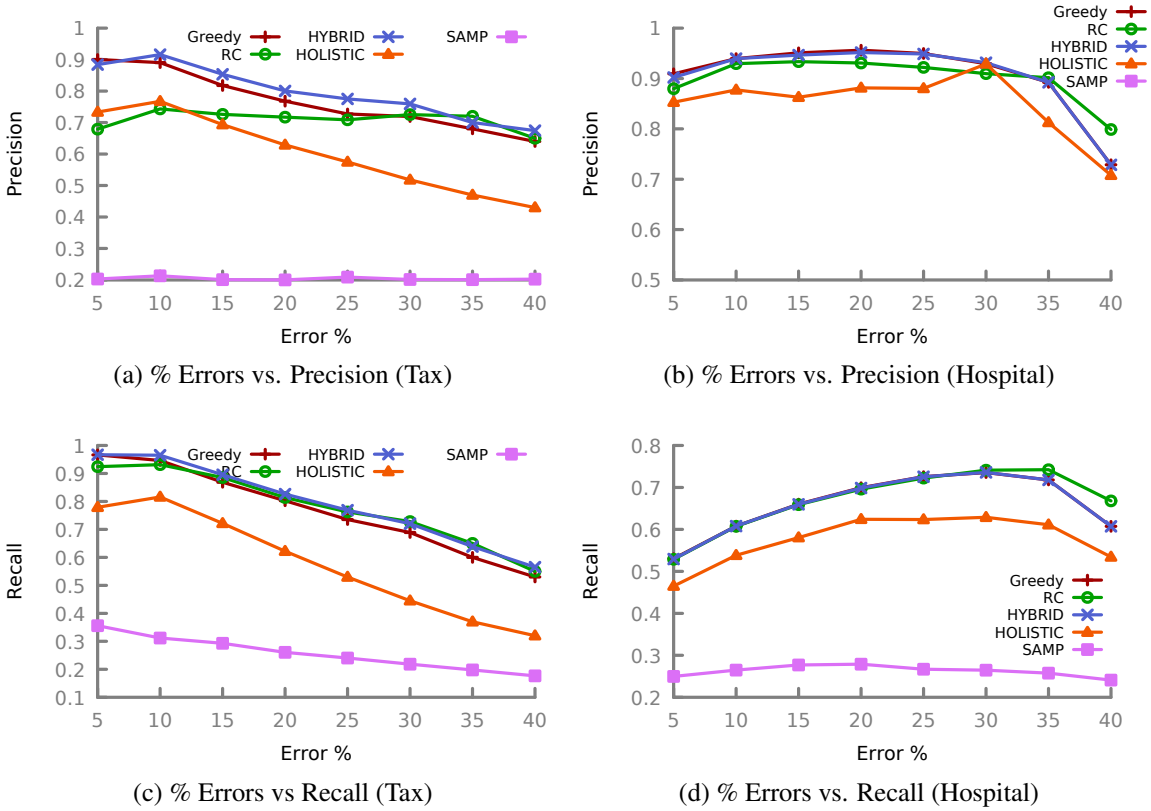
Figure 3.6.: Precision and Recall vs. #tuples

Error generation. We use BART [63] to benchmark different repair algorithms. BART makes it possible to introduce synthetic errors to the data so as to trigger violations of their corresponding FDs. For each dataset, we generate errors for all the defined FDs with varying noise levels and report the quality of the evaluated repair algorithms. We vary the size of the datasets, and for each size, we generate errors to violate the FDs.

Algorithms. We evaluate the following repair algorithms:

- *Greedy, RC and Hybrid*: We evaluate the three variants of our algorithm (as outlined in Section 3.6) and contrast their performance on different data sizes and error rates. As a notation, GRH refers to all the three variants.

- *Holistic* [16]: This is the state-of-the-art rule-based repairing algorithm. As reported in [16], this algorithm, though designed for Denial Constraints, performs better than



(a) % Errors vs. Precision (Tax)

(b) % Errors vs. Precision (Hospital)

(c) % Errors vs Recall (Tax)

(d) % Errors vs. Recall (Hospital)

Figure 3.7.: Precision and Recall vs. %Errors

other FD repair algorithms as it takes advantage of the interaction between the violations of different integrity constraints to achieve a minimal repair.

- *SAMP* [59]: This repairing algorithm computes FD (and CFD) repairs. The technique relies on sampling from different possible repairs, and may produce a different repair at each run. In order to be fair to this technique, we report the best repair quality results over 5 different runs.

- *TREP* [64]: This is a recent FD repair algorithm that computes string similarity to decide, given a threshold, if two tuples violate a given FD.

Metrics. We consider the traditional metrics to evaluate the quality of the produced repair instances: (1) Precision: The number of correct cells over the total number of changed

cells; (2) Recall: The number of correct cells over the total number of dirty cells.We also evaluate the runtime for different sizes of the data.

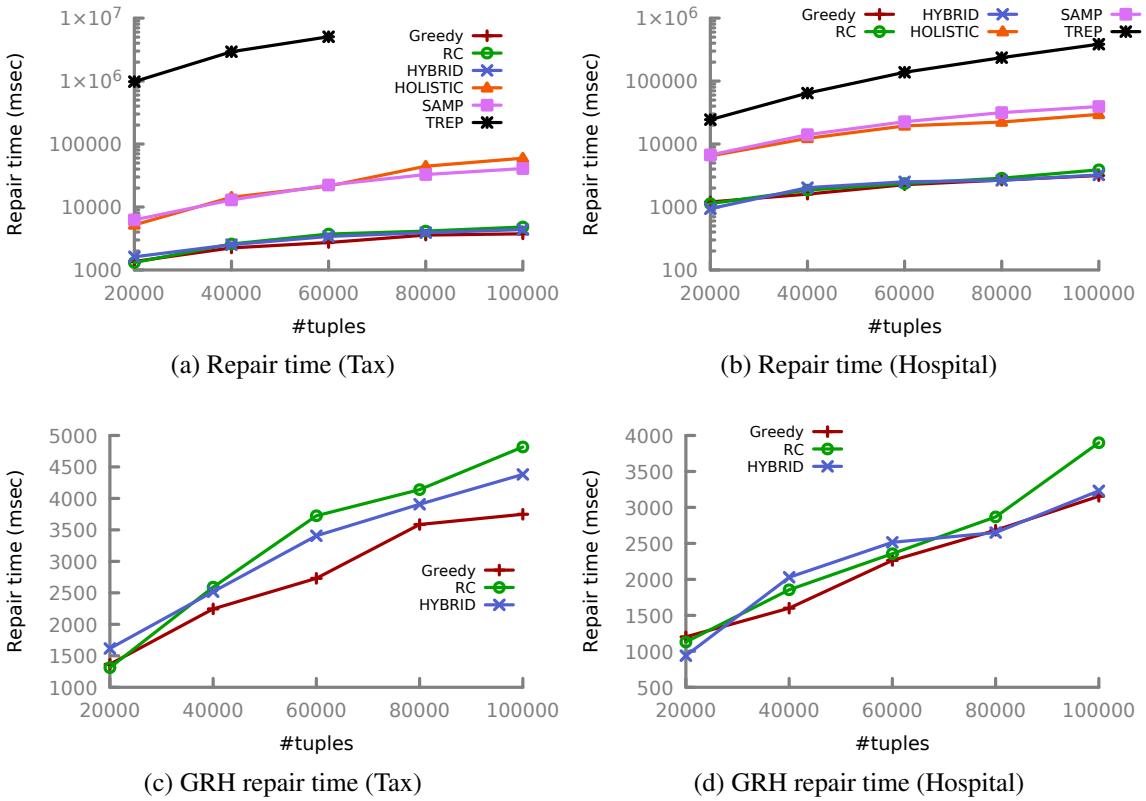
Implementation and Hardware Platform. All the algorithms are implemented in Java. All the experiments are conducted on a Linux machine with 8 Intel Xeon E5450 3.0GHz cores and 32GB of main memory.

3.7.2 Effectiveness Results

Figures 3.6(a-d) show the precision and recall on the *Tax* and *Hospital* datasets when the number of input tuples is varied for the various repair algorithms. In general, the three variants of our algorithm outperform the other baselines with the exception of *Holistic* in the case of precision for one of the datasets. Figure 3.6(a-b) illustrate that the precision of our algorithms is generally stable over different data sizes. In the *Hospital* dataset, *Holistic* and *GRH* have a comparable precision and recall, whereas in the *Tax* dataset, *GRH* performs much better. *SAMP* and *TREP* produce repairs with low precision and recall. The reason is that the former makes minimal random changes to the data to produce a repair (a consistent instance w.r.t. the FDs) while the latter relies on high similarity of FD attribute values to produce good repairs. Notice that *TREP* does not terminate for some data sizes (80K and 100K on Tax) as it runs out of memory (32 GB main memory).

From the results in Figure 3.6(a-d), observe that *Hybrid* reports the most consistent precision and recall values. The reason is that sometimes it is better to select the next FD pattern greedily (when it has a quality value above a given threshold). However, when the adjacent FD patterns have low quality, it is more beneficial to perform more careful pattern selection by computing the repair cover of the adjacent nodes before selecting an FD pattern. The threshold for *Hybrid* is set to 0.5, i.e., *Greedy* is used when an edge quality exceeds 0.5, otherwise *RC* is used.

Holistic performs poorly on the *Tax* dataset compared to the *Hospital* dataset. To guarantee termination, *Holistic* assigns *fresh values* when it cannot assign a value that eliminates the violations. We notice that the number of introduced *fresh values* is significantly higher in the *Tax* dataset compared to the *Hospital* dataset.



(a) Repair time (Tax)

(b) Repair time (Hospital)

(c) GRH repair time (Tax)

(d) GRH repair time (Hospital)

Figure 3.8.: Runtime results on *Tax* and *Hospital*

The rate of data errors affect all algorithms. That is, the more the errors the less evidence there is to correctly repair the data. Figures 3.7(a-d) report the precision of the produced repairs w.r.t. different data error-rates. Our algorithms clearly outperform the other systems, especially when there are more errors in the data. The reason is that adding more errors to data makes it harder for minimality-based algorithms to identify correct cell values, whereas in *GRH*, we go beyond the attribute-level when selecting a repair value; we select values that lead to the most supported FD patterns.

Greedy performs well when the error-rate is small (Figure 3.7(a-d)). However, as the error-rate increases, *Greedy* is outperformed by *RC* and *Hybrid*. The reason is that *Greedy* performs best when the FD patterns adjacent to the LHS nodes in the instance graph are most likely correct, but this changes as the error-rate increases, and a more careful traversal of the instance graph (as in *RC* and *Hybrid*) is needed for best results.

3.7.3 Runtime Results

We report the runtime results in Figures 3.8a and 3.8b for the *Tax* and *Hospital* datasets respectively. Our algorithms significantly outperform *Holistic* and *SAMP* by an order of magnitude, and *TREP* by three orders of magnitude when the data size is 100K. This is not surprising as our algorithm does not perform the detection step typically used in data repairing algorithms. This step is usually the most costly part of repairing algorithms. Since our algorithms run linearly in the number of FDs and tuples, the repairing time grows linearly as the data size increases.

We report the runtime of different variations of our algorithm in Figure 3.8c and 3.8d for the *Tax* and *Hospital* datasets, respectively. We observe that *RC* takes the longest to run compared to *Greedy* and *Hybrid*. This is expected as *RC* computes repair covers of every node before selecting FD patterns. *Hybrid* takes less time to run but has a higher running time than *Greedy*. In general the difference in running time between our algorithms is not significant.

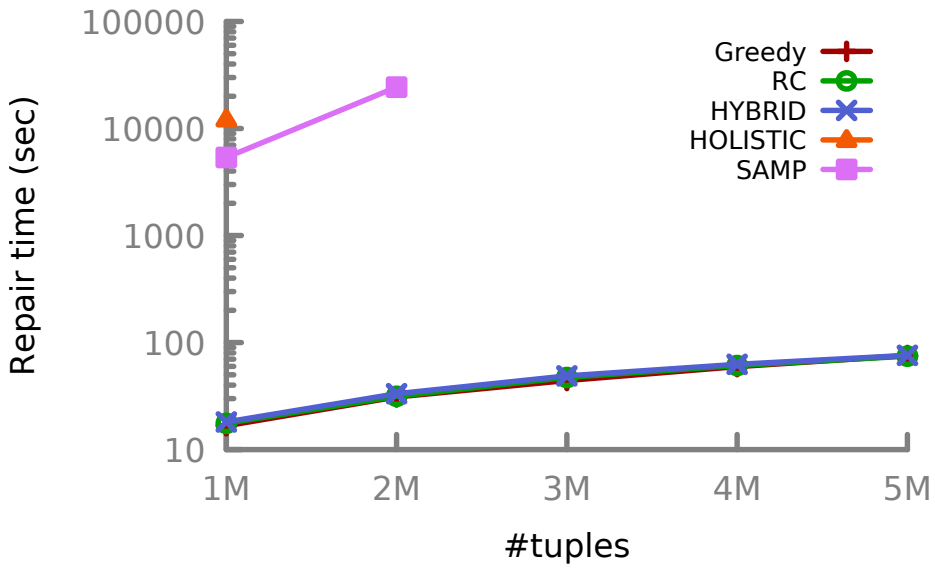


Figure 3.9.: Repair time on Tax_Extended

We report runtime results on *Tax_Extended* in Figure 3.9. Missing data points indicate that the algorithm does not finish after 24 hours. In the 1-million dataset, *GRH* outperforms *Holistic* and *SAMP* by three and two orders of magnitude, respectively. The reason is that *Holistic* and *SAMP* focus on finding minimal repairs to the data, which is typically a slow process. Due to its linear-time algorithm, *GRH* scales very well in larger datasets (it takes 75 seconds to clean a 5-million dataset).

3.8 Related work

There is a plethora of research on data cleaning [7]. Rule-based data cleaning techniques are the most related to our work as they take the same input as our algorithms, and do not assume the presence of external sources of clean data (master data) or humans to aid the repair process. Similar to our work, rule-based techniques output a database that is consistent with the defined rules. In a broader spectrum, our work is related to data cleaning efforts that may or may not use rules to derive their repairs, as well as those that benefit from the user's feedback. There is also a body of research on pattern discovery in the data for the purpose of deriving interesting rules to clean the data.

Existing rule-based data repairing techniques focus on computing repairs that change the database minimally to satisfy a set of rules, e.g., FDs [17, 18]. Conditional Functional Dependencies [65], Denial Constraints [16]. A wide array of techniques have been proposed to repair the data by modifying it minimally. Our work provides a significant addition to this family of repair algorithms from the way we model the data (FD patterns) to the way we present it to the user (repair expressions). Furthermore, unlike existing rule-based solutions, our work benefits from evidence from all the data values, including those that are not involved in violations to compute repairs.

In [58], probabilistic inference is employed to produce repairs based on different signals (constraint violations, external data, etc.). The produced repairs are associated with marginal probabilities that reflect their accuracy. The Algorithm proposes different sets of repairs that can then be validated by the user. Our work is different than [58] in two

ways: (1) unlike [58] we do not treat error detection as a black box, which makes the repair decisions highly influenced by the error detection approach used; (2) we produce "exact" repairs instead of probabilistic repairs that have to be processed by the user. Another probabilistic approach [57] relies on prediction of attribute values given the data distribution. Unlike our work, since the technique in [57] does not involve data quality rules, it does not produce an instance that is consistent w.r.t. to any defined rules (even if they are available).

Another related line of research focuses on discovering patterns in the data to build rules, which are then used to detect errors in the data [66]. In [66], the authors propose a technique to discover patterns that are then used as CFDs to enforce on the data. An attribute lattice is computed to explore different combinations of attributes and evaluate the interestingness of their value combinations. In our work, we discover the patterns that are induced from the interaction of FDs to repair the data and not to discover rules.

3.9 Concluding Remarks

In this chapter, we presented a novel repair approach that is a radical departure from most existing repair approaches. Guided by functional dependencies on the data, our proposal aims to generate a set of data modifications that exploit inherent patterns found in the data, in the form of value combinations based on the functional dependencies, to produce an accurate repair instance. Additionally, we compute the repair instance in linear time and produce pattern expressions that can easily be consumed by a human to understand the rationale behind the data modifications.

4 QUERY-TIME FUNCTIONAL DEPENDENCY REPAIRING

In this chapter, we present techniques to perform FD repairs at query time. As seen in Chapter 3, using pattern-preserving repairs significantly improves the quality of the repaired data. We adapt our previously presented solution to produce pattern-preserving repairs in the online setting, i.e., given a query result and a set of FD rules, we would like to repair the query result with respect to the defined FD rules. To achieve this goal, we present in this chapter adaptations to the techniques presented in Chapter 3 to support the online setting.

This chapter is organized as follows. We discuss related work in Section 4.2. Section 4.3 presents the problem definition and we present our solution in Section 4.4. We present metrics for FD patterns in Section 4.5. Section 4.6 presents our query-time FD repairing algorithm. We present our experimental results in Section 4.7 and conclude in Section 4.8.

4.1 Introduction

As we collect massive data from several sources, it is often the case that we want to clean the integrated data. This is because data integration is especially prone to introducing errors in the integrated instance [37, 67, 68]. In many applications, access to the underlying sources is restricted through query interfaces only, e.g., deep Web [19, 20]. In this chapter, we present techniques to enforce a set of FDs on query answers integrated from multiple Web sources. Particularly, we present a proof of concept adaptation of our techniques presented in Chapter 3 to work in the online setting. This chapter is a direct continuation of Chapter 3, therefore, the reader should refer to Chapter 3 prior to reading this chapter. As in Chapter 2, we use a virtual integration system (VIS) as an example application. The VIS integrates query results from several Web databases at query time.

Consider the example two query results in Table 4.1 and the following FDs: ϕ_1 : $Phone \rightarrow Zip$ and ϕ_2 : $Zip \rightarrow State$. As was observed in the case of online record linkage and fusion (Chapter 2), looking at query results in isolation to clean them is not effective. For instance, in Table 4.1, Q_2's answer contains a violation of ϕ_1, however, without looking at Q_1's answer, existing FD repair algorithms would choose either one of the Zip code values in $t_4[Zip]$ or $t_5[Zip]$ to lift the violation. A better way to compute the repair would be to repair Q_2's answer jointly with Q_1's answer. This allows gathering more evidence to compute the repair. Furthermore, sometimes records would not even trigger a violation when they appear in a query result, but, looking at them jointly with other query result sets would. For instance, we would not even know that records t_1, t_2 and t_6 are in violation of ϕ_2 if we do not repair Q_2 jointly with Q_1. We outline our contributions as follows:

- We propose a technique to perform qualitative FD repairs at query time and for multiple sources. To the best of our knowledge, this is the first time this has been addressed.
- We propose a model to represent query results as instance graphs, that are then cached for future reference.
- We develop metrics to measure the quality of FD patterns coming from several sources.
- We propose a model to incrementally update the quality scores of cached FD patterns

4.2 Related Work

There is a plethora of proposals that fall in the area of "rule-based data cleaning". Previous work that relates to FD repairing in the offline setting has been discussed in Chapter 3. In this section, we focus on previous proposals that deal with iterative and query-time integrity constraint repairing.

Table 4.1.: Sample query results on multiple Web sources

	Source	Name	State	Zip	Phone
Query Q1					
t_1	S_1	Bacino's Of Lincoln Park	IL	60615	(773) 472-7400
t_2	S_2	Hyde Park Bbq & Bakery	IL	60615	(773) 330-0440
t_3	S_3	Caffe Deluca Forest Park Inc	IL	60130	(708) 366-9200
Query Q2					
t_4	S_1	Caffe Deluca Forest Park Inc	IL	60130	(708) 366-9200
t_5	S_2	Caffe Deluca Forest Park Inc	IL	94016	(708) 366-9200
t_6	S_3	Hyde Park Bbq & Bakery	NY	60615	(773) 330-0440

Returning query answers that are consistent with respect to integrity constraints has been tackled in a research topic known as *Consistent Query Answering* [69]. The goal of *Consistent Query Answering* is to produce a query answer that is consistent with respect to a set of integrity constraints [69–73]. The idea of a consistent query answer is this: given the set of all possible repairs R for a query Q, produce a repair that is true in all the repairs in R. This is achieved through tuple deletion [71] or updates to existing tuples [73]. Most of the existing approaches adopt a query-rewriting strategy to capture all the possible repairs for a given query [69, 70]. This line of research focuses on the query structure and different approaches deal with different classes of queries. Our work does not depend on the query structure since we do not adopt query-rewriting as a strategy to generate repairs. Moreover, efforts in *Consistent Query Answering* solely focus on generating query answers that are consistent with respect to the integrity constraints, without considering the quality of the generated repairs. This is different from our proposal where consistency and quality of repairs are both considered.

The proposal in [22] studies the problem of FD repairing in the iterative setting. That is, the data and the rules are assumed to change over time and the system identifies repairs that need to be recomputed. A classifier is trained by a human and then applied to predict the type of repair to perform, i.e., change the data, the FDs, or both, given an FD violation. Our proposal is different from this system as we do not require a human intervention as

part of the cleaning algorithm. Moreover, in our setting, we are not only considering the iterative setting, but also the online setting, in which repair time is critical.

4.3 Terminology and Problem Statement

Let Q be a query posted to $S = \{S_1, S_2, ..., S_n\}$ relational tables and let $I = \{I_1, I_2, ..., I_n\}$ be the lists of records returned by each source table where I_i is the list of records returned by source S_i ($i < n$).

The union of Q's results is mapped to a global target table with schema T. The mapped instance is denoted by I_Q. Let Σ be the set of functional dependencies defined over T. As in Chapter 3, we assume that Σ is minimal and is in canonical form [60]. We aim to compute a repair for I_Q, denoted $Repaired(I_Q)$, that is consistent with Σ (denoted by $Repaired(I_Q) \models \Sigma$). Additionally, we denote by $S(X \rightarrow Y, x, y)$ the list of sources that returned the simple FD pattern $P(X \rightarrow Y, x \rightarrow y)$ for a given query Q.

Let $S^{\cup}$ be the union instance containing all the records in the source tables in S (we assume every record has a unique identifier and no two records can share the same identifier). That is, $S^{\cup} = S_1 \cup S_2 \cup ...S_n$. Let S^* be a pattern-preserving repair of $S^{\cup}$. Let I_Q^* be the set of records in S^* where $\forall t \in I_Q, t.id \in I_Q^*$ and I_Q^* contains only the records in I_Q.

Our goal is to: (1) enforce Σ over I_Q; and (2) minimize the distance between $Repaired(I_Q)$ (query-time repair of I_Q) and I_Q^* (query result cleaned by looking at the entirety of the data). The distance between two instances is the number of cell values that differ between them. Minimizing the distance between the cleaned instance and a reference instance has been widely used in the literature as a cost model for data cleaning algorithms [7, 16, 17]. Intuitively, we would like to produce query-time pattern-preserving repairs that are as close as possible to pattern-preserving repairs we would get if we had all the data.

4.4 Solution Overview

In Chapter 3, we showed that our proposed techniques work well to clean the data in one shot, and when all the data is available. However, in the online setting, where we perform cleaning at query time, data comes in small chunks, produced by user-posted queries. This calls for key changes in our proposed offline techniques (Chapter 3) to support the online setting. We outline those changes as follows:

1. Lack of global data view: Since the data is not available in its entirety, we need a way to collect relevant data to help us clean query results. Cleaning query results in isolation is bound to produce poor quality results. We maintain a cache that we update every time we process a new query, i.e., the cache contains FD patterns from previously processed query results. This cache will then be used to repair the query results jointly with previously processed query results.

2. Cache updates: Updating the cache has to be performed carefully as new query results could trigger updates to a large portion of the cache instance graph. The quality scores (as presented in Chapter 3) have to be updated every time we update the edges in the cache instance graph. In Section 4.5, we present two strategies to address this challenge.

Figure 4.1 outlines the architecture of our system. After a query Q is posted to the VIS, the system generates the FD patterns P_Q for Q using FDs in Σ. Those patterns are then appended to the cache Instance Graph IG containing FD patterns from previously processed queries. The update in the Instance Graph incurs an update to the quality scores of FD patterns affected by P_Q. We will outline different strategies to update the quality scores in Section 4.5. Finally, the system generates the repair expressions which constitute the cleaned query answer. The static analysis part (highlighted with a dashed rectangle in Figure 4.1) generates the FD graph, orders the FDs and generates a traversal order. Those steps are identical to the ones presented in Chapter 3.

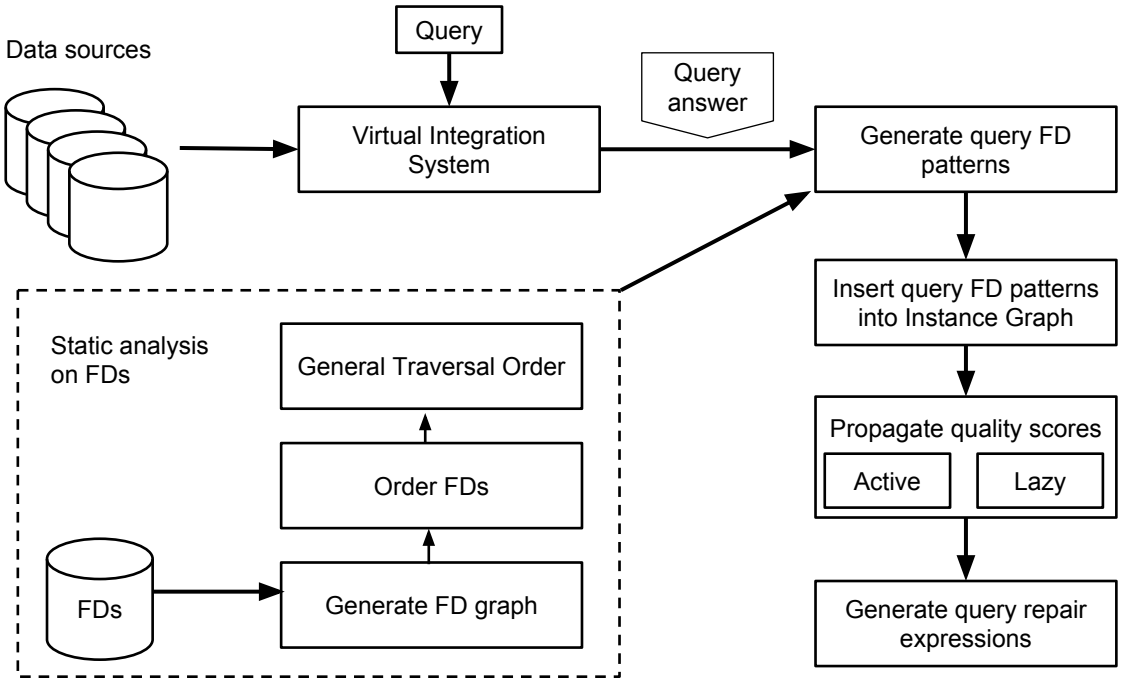


Figure 4.1.: Architecture for query-time FD repairing

4.4.1 Iterative Caching of Functional Dependency Patterns

Since we do not assume we have access to the entirety of the data, we build a cache from posted queries and clean incoming queries jointly with the cache. We employ a similar philosophy as in our previous work [20], where we used iterative caching to perform query-time record linkage and fusion. In the absence of a "global" view on the data, it is not realistic to expect any cleaning algorithm to clean the data reasonably, i.e., cleaning algorithms cannot "invent" data that was not even part of the input data. We aim to address this problem by iteratively constructing a cache that contains the simple FD patterns of previously processed queries. We first construct the instance graph of query result I_Q which only contains the simple FD patterns seen in I_Q. We refer to this graph as the Query Instance Graph, or IG_Q for short. The cache is modelled as a graph of simple FD patterns, i.e., it contains all the FD patterns that were processed in previously posted queries, we refer to this graph as the Global Instance Graph (IG), or simply the cache.

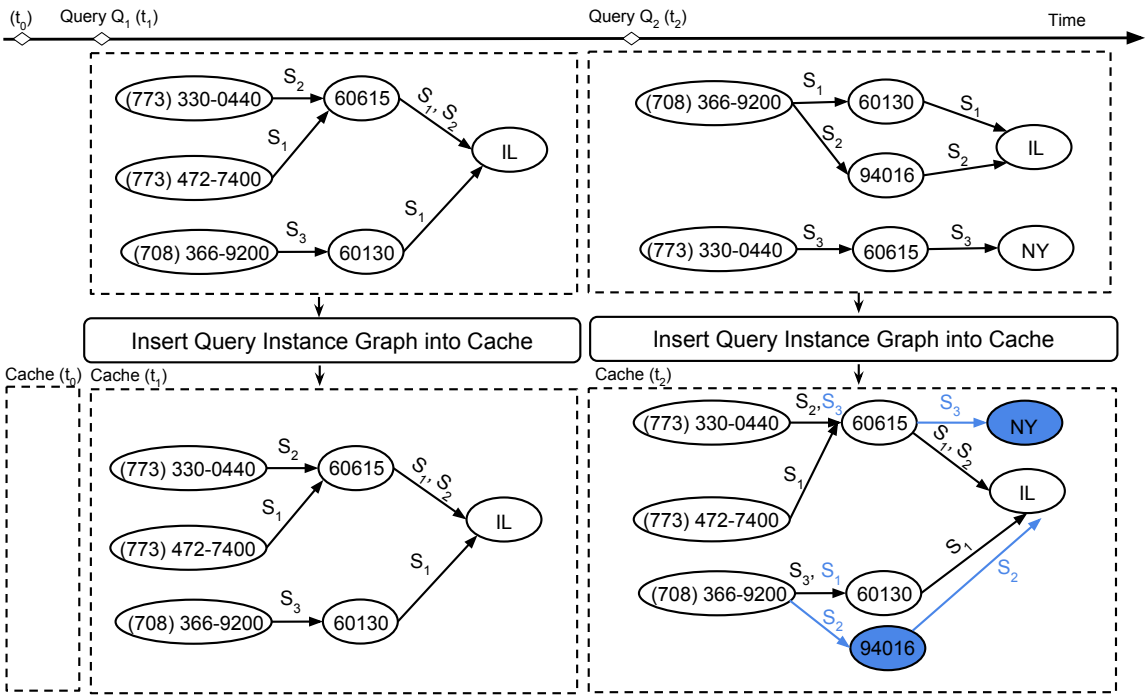


Figure 4.2.: Example query instance graphs being appended to the cache over time

Appending the Query Instance Graph into the Cache

Our system processes query results one at a time. Every query result I_Q is inserted to the cache before answering the query Q. We define the union of $IG_Q(V_Q, E_Q, S_Q)$ and $IG(V, E, S)$ such that V_Q, E_Q and S_Q are the set of vertices, edges and sources in the query instance graph, respectively, and V, E and S are the set of vertices, edges and sources in the cache. Appending I_Q to IG (denoted $I_Q \oplus IG$) is defined as follows:

$$IG_Q(V_Q, E_Q, S_Q) \oplus IG(V, E, S) = G(V_Q \cup V, E_Q \cup E, S_Q \cup S) \tag{4.1}$$

Figure 4.2 illustrates an example timeline (on the top in Figure 4.2) with two queries posted Q_1 and Q_2 (from Table 4.1) at times t_1 and t_2. Initially, the cache is empty at time t_0. At time t_1, a query Q_1 is posted. Q_1's Instance Graph is inserted into the cache. At time t_2, query Q_2 is posted and its Instance Graph is inserted into the cache. The parts that are coloured in blue are the ones that were inserted as a result of processing Q_2. Each edge

in the graphs embeds the set of sources that provided that edge, which will be required to compute the quality scores of the edges (Section 4.5).

4.5 Quality Metrics

Contrary to the setup in Chapter 3, we now have multiple data sources. We would like to harness the source provenance of records to characterize the quality of FD patterns. We modify the measures presented in Chapter 3 to express the quality of an FD pattern as a function of the number of data sources that provide it.

For a simple FD pattern $P : (X \rightarrow Y, [x, y])$, the $Support$ of P, denote $Sup(P)$, is the number of sources with $X = x$ and $Y = y$ in query result instance I over the total number of sources $|S(X \rightarrow Y, x, *)|$.

$$Sup(P) = \frac{|S(X \rightarrow Y, x, y)|}{|S(X \rightarrow Y, x, *)|} \tag{4.2}$$

* denotes "any value". $|S(X \rightarrow Y, [x, *])|$ denotes the number of sources in I with the LHS value x and any RHS value.

As shown in Chapter 3, quantifying the quality of a simple FD pattern P should also be done with respect to the set of simple FD patterns it leads to, denoted by $P^{\rightarrow}$. This is done by "propagating" the support values computed for all the FD patterns in $P^{\rightarrow}$ back to P. We define $Score(P)$ and $Quality(P)$ as follows:

$$Score(P) = Sup(P) + \sum_{Q \in P^{\rightarrow}} Sup(Q) \tag{4.3}$$

$$Quality(P) = \frac{Score(P)}{|P^{\rightarrow}| + 1} \tag{4.4}$$

4.5.1 Propagating the Quality Scores in the Instance Graph

In Chapter 3 (offline setting), we computed the quality of all FD patterns at once. However, in the online and iterative settings, queries cause the data to come in chunks and

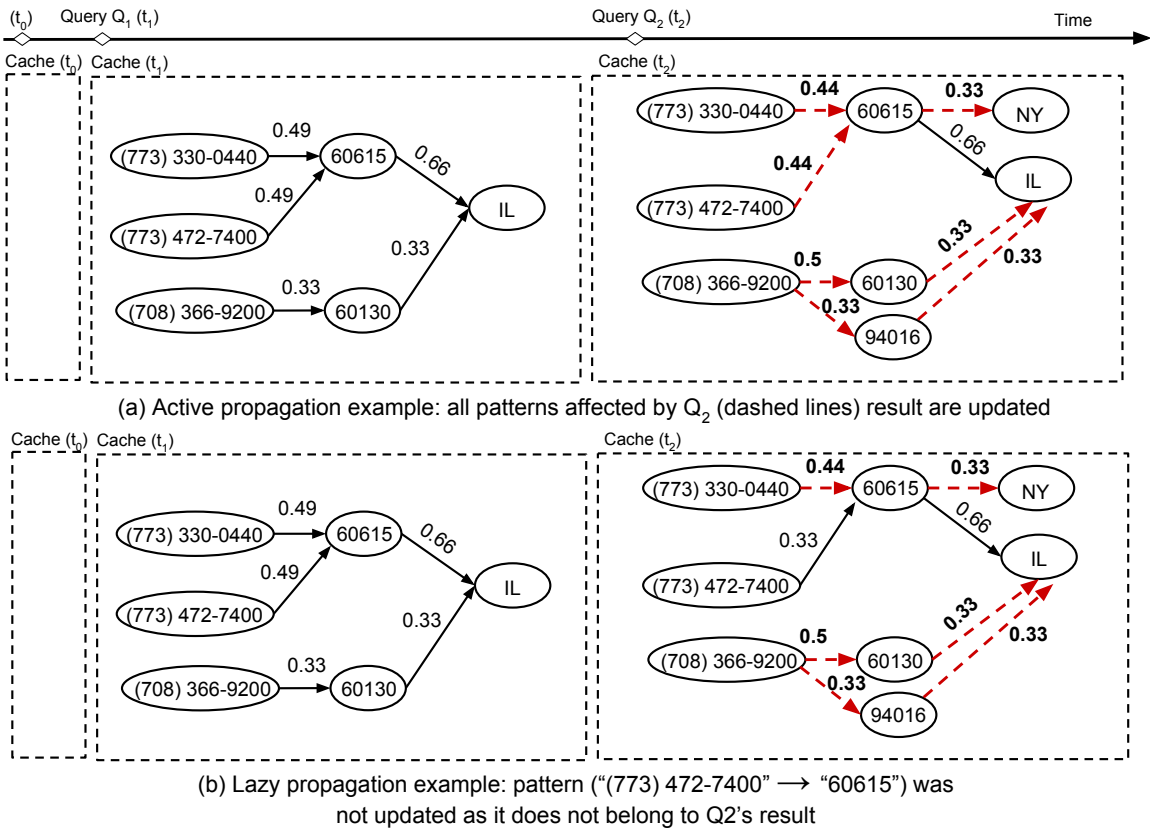


Figure 4.3.: Score propagation at times t_1 and t_2: Q_2 resulted in propagating the scores in the cache, dashed lines represent patterns whose scores were updated, boldface scores represent scores that were affected by the propagation

not all at once. As a result, $Quality(P)$ has be computed incrementally when the data is updated, i.e., we should only propagate the support values for the FD patterns in $P^{\rightarrow}$. Particularly, we propose two strategies to propagate the scores in the instance graph at query time: (1) **Active propagation:** Given a query result I_Q, we propagate the scores of the FD patterns in I_Q to all the FD patterns they can lead to in the cache; and (2) **Lazy propagation:** Given a query result I_Q, we propagate the scores of the FD patterns in I_Q to only the FD patterns in the query instance graph IG_Q.

Active Propagation

Given the simple FD patterns P_Q extracted from a query result I_Q, we update all the simple FD patterns P in IG where: (1) P contains all the simple FD patterns in IG where each simple FD pattern $p \in P$ is reachable from at least one simple FD pattern $q \in P_Q$; (2) we recompute the quality scores for all the simple FD patterns in P and P_Q. The worst-case time complexity of active propagation is $O(|V|+|E|)$ where V and E are the set of vertices and edges, respectively in the cache instance graph. That is, it is possible that a query result triggers updates to all the edges in the cache instance graph. This is because we follow any edges that the edges in P_Q lead to in the cache.

Example 4.5.1 *Consider the two query results in Table 4.1. Assume, the two queries Q_1 and Q_2 are posted at times t_1 and t_2 respectively. Figure 4.3(a) illustrates the state of the cache after processing both queries using active propagation. At time t_1, Q_2's query result has resulted in updating seven simple FD patterns (dashed lines). The scores are recomputed for all the edges that were affected by Q_2's results. To show an example calculation of the quality scores, consider the following simple FD patterns at time t_2 in Figure 4.3(a):*

- $P_1 : (\phi_2,$ "60615", "NY")
- $P_2 : (\phi_2,$ "60615","IL")
- $P_3 : (\phi_1,$ "$(765)\ 330-0440$", "60615")
- $P_4 : (\phi_1,$ "$(765)\ 472-7400$", "60615")

The quality of P_3 with $P_3^{\rightarrow} = \{P_1, P_2\}$ is calculated as follows: $Quality(P_3) = \frac{Sup(P_3)+Sup(P_1)+Sup(P_2)}{|P_3^{\rightarrow}|+1} = \frac{0.33+0.33+0.66}{3} = 0.44$

Lazy Propagation

Given the simple FD patterns P_Q extracted from a query result I_Q, we update only the simple FD patterns P in IG where P contains the simple FD patterns in IG where

$P = P_Q$. That is, we only update the scores of the query's simple FD patterns. In this approach, we may have simple FD patterns in the cache that have "incomplete" quality scores, i.e., those FD patterns will only be updated if they appear as part of future query results. The worst-case time complexity of this strategy is $O(|V_Q| + |E_Q|)$, where V_Q and E_Q are the set of vertices and edges in the query instance graph. Since this latter is generally significantly smaller than the cache graph, this strategy is suitable in cases where the cache graph becomes very large and active propagation becomes a big overhead to the query response time.

Example 4.5.2 *Refer to Figure 4.3(b). The dashed lines highlight the FD patterns that were updated at t_2. Note that as opposed to the case of active propagation (Figure 4.3(a)), $P_4 : (\phi_1, \text{“}(765)472 - 7400\text{”}, \text{“}60615\text{”})$ was not updated. This is because P_4 was not part of Q_2's answer. Note that with lazy propagation, the FD patterns that were affected as a result of processing Q_2, and are not in Q_2's answer, are updated only when another query returns those FD patterns. For example, the quality score of P_4 will only be updated if a future query returns P_4 in its answer.*

Algorithm 8 performs the propagation of quality scores at query time. Given a query result I_Q, we first insert its set of simple FD patterns P_Q into the Instance Graph IG (Line 3). We then do a Depth-First Search traversal of the Instance Graph (Line 7). The traversal routine is described in Algorithm 7: we first check if we are doing an Active or Lazy propagation (Line 1). In Active Propagation, we recompute the quality scores for all the edges that can be reached from a given vertex v in the Instance Graph IG ($v.adjacent(G)$). In the case of Lazy Propagation, we only recompute the quality scores for the edges in P_Q (those that are part of the query result) ($v.adjacent(P_Q)$). The rest of Algorithms 7 and 8 is the same as Algorithms 3 and 4 (Chapter 3).

4.6 Query-Time FD Repairs

We are now ready to present adaptations to Algorithm 6 (Chapter 3) to support query-time FD repairing. Algorithm 9, given a query Q, computes a repaired query answer (ex-

Algorithm 7: Traverse(Vertex v)

output: Average quality of edges starting from Input Vertex v

```
1  if Active Propagation then
2      W ← v.adjacent(G)
3  else if Lazy Propagation then
4      W ← v.adjacent(P_Q)
5  if W = ∅ then
6      vQuality[v] ← 0
7  forall w ∈ W do
8      Edge e = (v, w)
9      Score ← 0
10     if visited[w] = true then
11         BackEdges ← BackEdges ∪ e
12         return 0
13     else
14         visited[w] ← true
15         Score ← Sup(e) + Traverse(w)
16         Quality[e] ← Score / (|e→|+1)
17         vQuality[v] ← vQuality[v]+ Score
18 return vQuality[v]
```

Algorithm 8: ComputePatternsQuality(Instance Graph G, Query FD patterns P_Q)

output: Edge-weights reflecting the quality of the FD patterns

```
1  BackEdges ← ∅
2  V ← P_Q.vertices
3  G = G ⊕ P_Q
4  for i ← 0 to |V| do
5      visited[i] ← false
6  forall v ∈ V do
7      Traverse(v)
8  forall Edge e(v, w) ∈ BackEdges do
9      Quality[e] ← avg(Sup(e), vQuality[w])
```

pressed as a pattern expression). The Algorithm follows the same logic as Algorithm 6, except that it computes repairs per query (Line 5). After computing the query answer I_Q (Line 6), the set of simple FD patterns in I_Q is extracted (Line 7) and then appended to

the cache instance graph (Line 9) where the quality scores are propagated. After that, we simply use the instance graph to compute the repairs. The repairing steps (Lines 10-24) are the same used in the offline setting (refer to Chapter 3 for details).

Algorithm 9: QueryTimePatternPreservingRepairs(Σ, Query Stream QS, Data Sources S)

output: For every query in QS, return pattern expressions

```
1  SCCG ← BuildSCCGraph(Σ)
2  OC ← TopologicalSorting(SCCG)
3  Ordered_FDs ← OrderFDs(Σ, OC)
4  IG ← ∅
5  forall Query Q ∈ QS do
6      I_Q ← Query_Data_Sources(q, S)
7      P_Q ← Get_FD_Patterns(I_Q)
8      pattern_expressions ← ∅
9      ComputePatternsQuality(IG, P_Q)
10     forall Tuple t ∈ I_Q do
11         for i ← 0 to |Ordered_FDs| do
12             forall FD f ∈ Ordered_FDs[i] do
13                 Lval ← t[f.LHS]
14                 if Rtable(f).contains(Lval) then
15                     FDPattern p ← New FDPattern(f, Lval → Rtable(f).get(Lval))
16                     P^exp(t) ← P^exp(t) ▷ p
17                 else if f.RHS ∈ P^exp(t) then
18                     FDPattern p ← New FDPattern(f, Lval → GetAttributeValue(P^exp(t),
                         f.RHS))
19                     P^exp(t) ← P^exp(t) ▷ p
20                     Rtable(f).Add(Lval, GetAttributeValue(P^exp(t), f.RHS))
21                 else
22                     FDPattern p ← Edge_Selection(IG)
23                     P^exp(t) ← P^exp(t) ▷ p
24                     Rtable(f).Add(Lval, p.RHS)
25         pattern_expressions = pattern_expressions ∪ P^exp(t)
```

4.7 Experimental Study

We evaluate our query-time FD repairing algorithm on the real-world restaurant dataset that we described in Chapter 2. We considered different variants of the cache as well as the

used score propagation strategies to evaluate our algorithm. Experiment results show not only a gain in repair quality over a state-of-the-art repairing algorithm, but also a runtime gain in query answering time.

4.7.1 Dataset and Queries

We use the restaurant datasets that we crawled from nine Web sources (Chapter 2). From this dataset, we synthetically generate a query stream of 100K queries following the approach described in Chapter 2. We defined four FDs over a target schema in which we integrate query results posted to the nine sources. The data as it stands contains errors, so we did not add any noise to it.

4.7.2 Ground Truth

We do not have the ground truth for the restaurant data we crawled. We use random sampling to pick queries for which we generate the ground truth manually. We divide the query stream into five buckets (0-20K, 20K-40K, 40K-60K, 60K-80K, 80K-100K) and we select 50 random queries within each bucket. In total we have 250 queries for which we manually generate the ground truth.

4.7.3 Algorithms

In Chapter 3, we assessed the quality of our pattern-preserving repairing algorithms against several other data repairing algorithms. As our experiment results showed, *Holistic* [16] seems to be the most promising competitor to our repairing approach. Therefore, we adapted *Holistic* to support query-time data repairing by making it process one query at a time instead of processing all the data at once. Since *Holistic* does not use any caching mechanisms, every query result is cleaned in isolation.

In our setting, a dynamic cache can be updated by every query posted to the system. A static cache is loaded with a set of queries (e.g., 20K queries), and then, the cache becomes

static. Incoming queries on a static cache do not modify the cache. As we show in our results, a static cache can greatly improve the query response time and can be a good option especially if we have cache data that we know is clean (e.g., human-curated data).

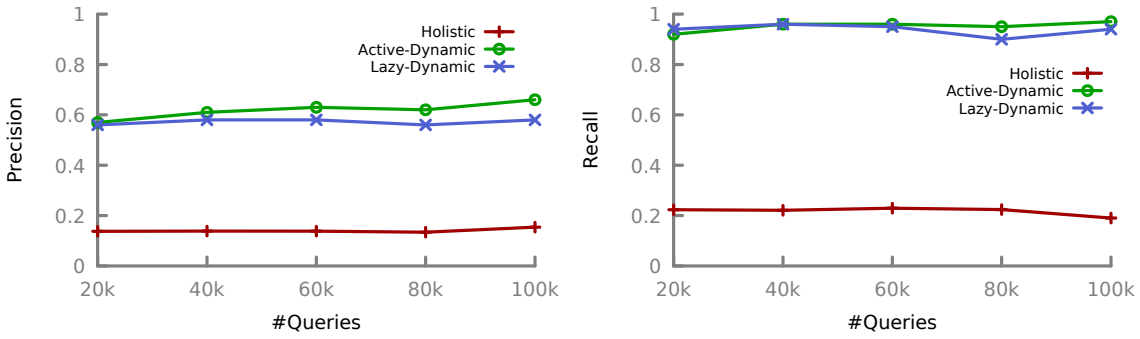
We have implemented three variants of our online algorithm: (1) Active-Dynamic: Greedy repair strategy with active score propagation and dynamic cache; (2) Lazy-Dynamic: Greedy repair strategy with lazy score propagation and dynamic cache; (3) Active-Static: Greedy repair strategy with active score propagation and static cache; and (4) Lazy-Static: Greedy repair strategy with lazy score propagation and static cache.

As experiment results showed in Chapter 3, the $Greedy$ variant of our algorithm provides reasonable quality compared to RC and $Hybrid$. Since $Greedy$ is more efficient than RC and $Hybrid$, we chose $Greedy$ to be the repairing strategy in the online version of the algorithm, i.e., in the online setting, we would like to strike a balance between the quality of query results and query response time.

Implementation and Hardware Platform. The algorithms are implemented in Java. The experiments are conducted on a Linux machine with 192 Intel Xeon Platinum 8168 2.70 GHz cores and 3 TB of main memory.

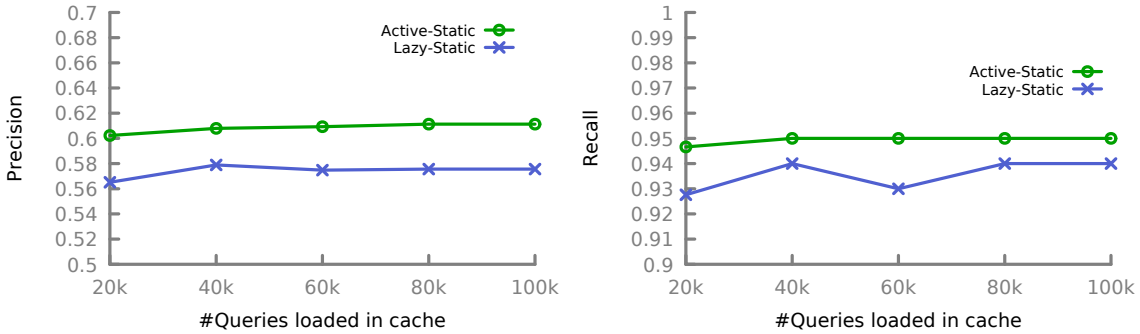
4.7.4 Metrics

We use Precision and Recall to measure the quality of query results. Let I_Q, $Repaired(I_Q)$ and $Ground(I_Q)$ be the raw query result, its repaired version (at query-time) and its ground truth, respectively. Moreover, if a cell was originally clean in I_Q and our repairing algorithm did not change it, this cell will not be considered in our computations of Precision and Recall. That is, $Repaired(I_Q)$ only contains cells that were changed in I_Q and not initially clean. Similarly, $Ground(I_Q)$ only contains cells that are not initially clean in I_Q and I_Q contains cells that are not clean. Precision for a query Q is defined as the ratio of correctly repaired cells ($|Repaired(I_Q) \cap Ground(I_Q)|$) over the total number of cells in the query result ($|I_Q|$). Formally, precision for a query Q is defined as follows:



(a) #Queries vs. Precision (Dynamic)

(b) #Queries vs. Recall (Dynamic)



(c) #Queries vs. Precision (Static)

(d) #Queries vs. Recall (Static)

Figure 4.4.: Online FD repairing effectiveness results

$$Precision(I_Q) = \frac{|Repaired(I_Q) \cap Ground(I_Q)|}{|I_Q|} \tag{4.5}$$

Recall for a query Q is defined as the ratio of correctly repaired cells ($|Repaired(I_Q) \cap Ground(I_Q)|$) over the total number of cells in the ground truth of the query result ($|Ground(I_Q)|$). Formally, recall for a query Q is defined as follows:

$$Recall(I_Q) = \frac{|Repaired(I_Q) \cap Ground(I_Q)|}{|Ground(I_Q)|} \tag{4.6}$$

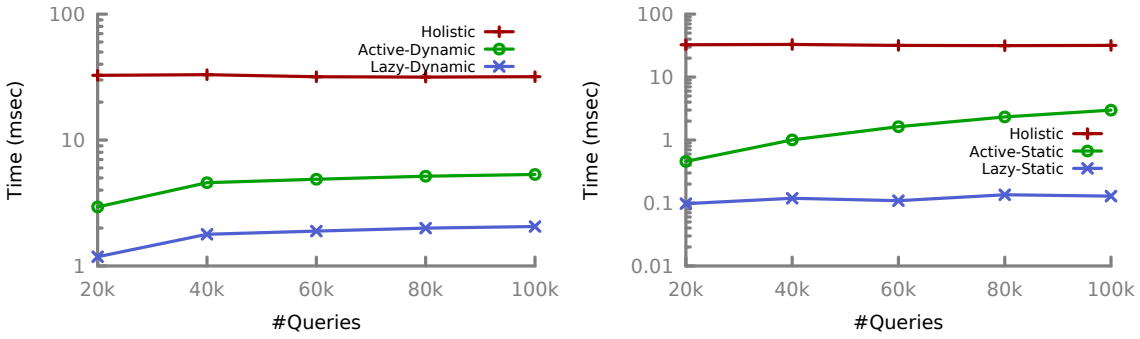
We compute the precision and recall for the randomly picked 50 queries and we average the result.

4.7.5 Results

Figures 4.4a and 4.4b report the precision and recall results of *Active-Dynamic*, *Active-Lazy* and *Holistic*. We observe that *Active-Dynamic* and *Lazy-Dynamic* significantly outperform *Holistic* by up to 5 times. We can also see that the recall results for *Active-Dynamic* and *Lazy-Dynamic* (Figure 4.4b) greatly outperform *Holistic*. This whopping difference in quality between our approach and *Holistic* is due to two factors: (1) In Holistic, every query result is cleaned in isolation, so, there is no "global view" over the data when repairing the query result; and (2) As shown in Chapter 3, pattern-preserving repairs are generally more accurate than repairs computed by $Holistic$, even when all the data is available. We can see that the active propagation strategy is generally superior to the lazy strategy, but only marginally (Figures 4.4a and 4.4b). This is because the posted queries were able to minimize the window during which the quality scores of FD patterns are not propagated. That is, eventually, there were queries that resulted in propagating the scores, and hence, rendering the FD pattern scores current. Though it performs marginally less in terms of quality than its active counterpart, the runtime of the lazy strategy is far less than the runtime of the active strategy (Figure 4.5a). This suggests that the lazy propagation strategy strikes a good balance between quality of produced repairs and runtime.

Interestingly, our query-time repair algorithm also performs significantly faster than *Holistic* (Figure 4.5a), even though we have to do score propagation for every received query. This is due to the smart ordered traversal of the instance graph and the incremental propagation of quality scores.

The static configuration of the cache produces good quality repairs that are comparable to those computed with the dynamic cache configuration (Figures 4.4c and 4.4d). At each number of posted queries n, we report the average precision and recall for the whole 100K query stream when the cache is loaded with n queries. As expected, loading more queries into the cache results in better quality. We also observe that Lazy-Static is more sensitive to cache size than Active-Static. This is because when building a cache with the lazy propagation strategy, we may be left with many pattern quality values that were not propagated yet.



(a) #Queries vs. Time (Dynamic)

(b) #Queries vs. Time (Static)

Figure 4.5.: Online FD repairing efficiency results

This is because the system may not have processed a query that would result in computing them. Therefore, the more queries we load into the cache, the more likely we get the scores propagated in the cache, and hence, produce better repairs.

Running the system with a static cache greatly improves runtime (Figure 4.5b). This is expected as the static configuration will avoid doing any score propagation or updates to the cache. A static cache is suitable when we have a subset of the data that has a high accuracy, e.g., curated by experts.

4.8 Concluding Remarks

We described in this chapter key adaptations to our offline FD repairing proposal to work in the online setting. We showed that our pattern-driven framework supports offline, iterative and online data cleaning settings. This is a significant step for data cleaning algorithms as we showed that we can use the same repairing framework to support different settings and not separate cleaning logics for each separate setting. Our experimental results show that our online FD repairing proposal can produce not only query answers with a reasonable accuracy, but also does so very efficiently, which is a critical requirement to support the online setting.

5 NEXT DIRECTIONS: HUMAN-DRIVEN DATA CLEANING

In this chapter, we present a vision for an important next direction in data cleaning: human-driven data cleaning, where humans and tools are both part of one data cleaning framework and can both participate in cleaning the data. Human involvement is instrumental at several stages of data cleaning, e.g., to identify and repair errors, to validate computed repairs, etc. There is currently a plethora of data cleaning algorithms addressing a wide range of data errors (e.g., detecting duplicates, violations of integrity constraints, missing values, etc.). Many of these algorithms involve a human in the loop, however, this latter is usually coupled to the underlying cleaning algorithms. There is currently no end-to-end data cleaning framework that systematically involves humans in the cleaning pipeline regardless of the underlying cleaning algorithms. In this paper, we highlight key challenges that need to be addressed to realize such a framework. We present a design vision and discuss scenarios that motivate the need for such a framework to judiciously assist humans in the cleaning process. Finally, we present directions to implement such a framework.

This chapter is organized as follows. We present the architecture of our envisioned system in Section 5.2. In Section 5.3, we discuss key features to characterize humans in the cleaning pipeline. In Section 5.4, we discuss the problem of automatically assigning humans to cleaning tasks. We present and contrast different cost optimization strategies in Section 5.5. We address strategies to identify bottlenecks in the data cleaning pipeline in Section 5.6. We discuss related work in Section 5.7 and conclude in Section 5.8

5.1 Introduction

Businesses often collect large volumes of data to inform key decisions. However, because data can be humongous and highly volatile, it is infeasible for humans to manually verify its accuracy. As a result, decision-makers have to deal with possibly-inaccurate data

that may inherently lead to faulty business decisions. There are abundant research efforts to detect and repair the many types of data errors that one sees in the wild. This process is also known as the data cleaning process. Data errors include duplicates [11], violations of integrity constraints [16], and missing values [74]. While ideally we want to be able to fully automate this process, it has been widely recognized that humans have to be involved at various stages of the data cleaning pipeline [7, 12, 75]. A large spectrum of data cleaning systems involve humans. Examples include Poter's Wheel [76], GDR [12], KATARA [77], CrowdER [78], and UGuide [9]. Each of these systems involves humans to solve a particular data cleaning task. However, an end-to-end data cleaning framework that involves humans in a way that is orthogonal to the underlying cleaning algorithms is not yet available. Looking at existing data cleaning techniques, we make the following observations:

Human involvement is algorithm-driven: Humans are the ultimate authority in verifying the accuracy of the data. Because it is impractical to have humans correct the entirety of the data, many techniques strive to involve humans judiciously so as to maximize the benefit of their feedback in the cleaning process [6, 12, 78, 79]. Typically, humans are tightly coupled to the cleaning logic, i.e., humans are involved in ways that are dictated by the cleaning algorithm being used. This coupling is necessary to produce good quality results for specific data cleaning tasks. However, if we want to "plug-and-play" arbitrary tools to clean different types of data errors, then, we need a way to involve and manage humans in an algorithm-agnostic fashion in the data cleaning pipeline. This generic inclusion of humans is not meant to replace human-guided algorithms, in fact, they should go hand-in-hand to unlock various use cases in the cleaning process.

Data singularity: An assumption that is usually made in cleaning algorithms is that the data is subjected in its entirety to a given cleaning algorithm [8, 58]. However, in practice, different parts of the data are cleaned by different agents. For instance, one may use an automatic data cleaning tool to find the correct mapping of the Zip code to a City name while requesting humans to correct the Salary data (one would not trust an automatic algorithm to modify the salary data). This motivates the need for a cleaning framework that supports

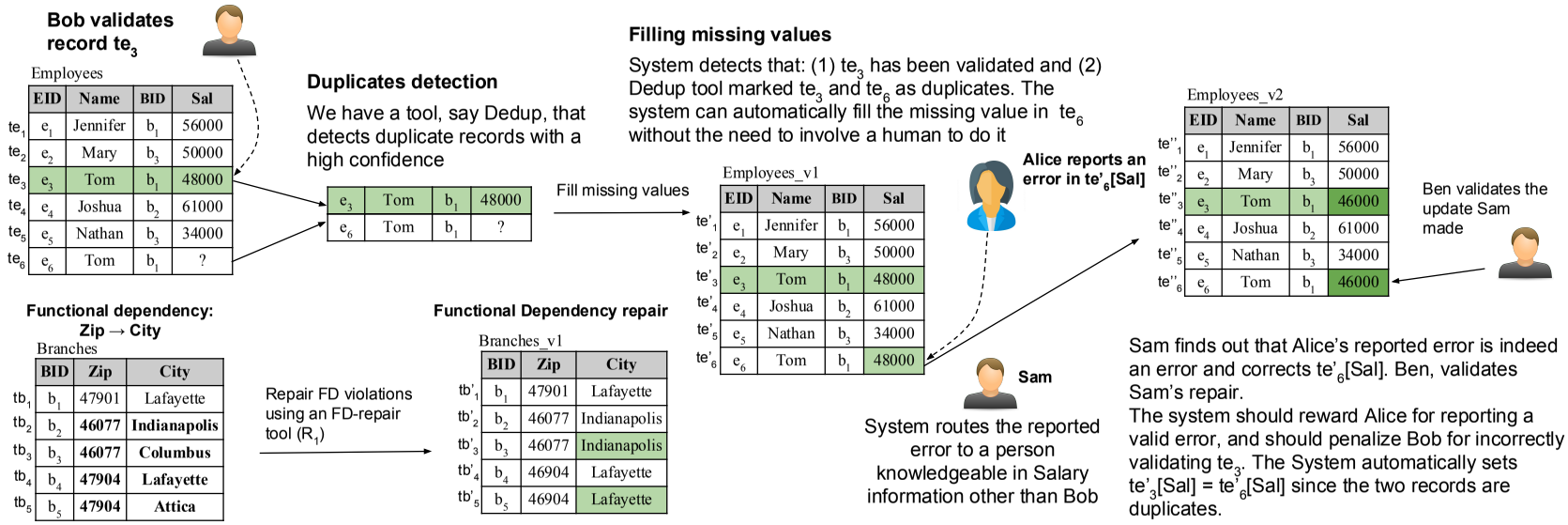


Figure 5.1.: Example data cleaning scenario involving various cleaning tasks and agents

both human and automatic agents to holistically clean different parts of the data. In the remainder of the chapter, we refer to the detection and repairing agents as *cleaning agents*.

Source of errors: There are multiple factors that can influence the quality of the computed repairs, e.g., the humans involved, the data quality rules, and the repair algorithm. However, existing techniques do not assess the effect of various factors that produce the data repairs (for example, many rule-based repair algorithms assume the rules are correct [17, 18, 65]). Understanding this effect is crucial in identifying bottlenecks in the data cleaning pipeline. Just as important as suggesting potential data errors for humans to verify, it is important to make it easy for humans to identify *faulty* factors (rules, humans, external resources, etc.) that have been involved to compute the inaccurate repairs.

Humans are not always right: Many human-driven data cleaning techniques assume that humans (e.g., experts) are *perfect* [12]. However, in practice, humans may make mistakes at various stages of the data cleaning pipeline (e.g., in the detection, repairing, or validation phases). Understanding how humans interact with the data is important to judiciously involve them in the cleaning process. For instance, an error reported in the *Sales* data by a person working in the *Sales* department should have more weight than one reported by a human working in another department. Therefore, there are several nuances in the human feedback that need to be dissected to effectively involve humans in the cleaning process.

Example 5.1.1 *Refer to Figure 5.1. Table Employees contains employee data, e.g., name, salary, and the branch they belong to (BID). Table Branches contains the list of branches (BID) and their location (Zip, City). We assume all branches are based in the State of Indiana, thus we omit the State attribute from Table Branches.*

Scenario 1: *As we illustrate in this example, there are many use cases where it is useful to have a high-level, algorithm-agnostic understanding of how different components in the cleaning pipeline interact with each other. Because they were designed to solve a specific data cleaning task, existing human-guided algorithms do not capture those use cases. In this scenario, we present a mix of data cleaning operations performed by various agents to repair the tables Employees and Branches. The workflow is as follows:*

1. *Bob validates record te_3 as being correct.*

2. *Table Employees is deduplicated using a tool, $Dedup$. This latter reports records te_3 and te_6 as duplicates.*

3. *We would like to fill in the missing values. Instead of assigning a human to do it, the system should be able to notice that since te_3 and te_6 are duplicates, and Bob previously marked record te_3 as being correct, then, the missing value $te_6[Sal]$ can be set to $te_3[Sal]$. The resulting table is $Employees\_v_1$.*

4. *Alice reports an error in cell $te'_6[Sal]$ in table $Employees\_v_1$.*

5. *The system should be able to automatically ask a human, Sam, who is knowledgeable about the Salary data to repair the reported error. The system should be able to notice that it is better to choose a human other than Bob to examine the error, so that it can then compare their decisions. Sam then corrects the error and updates $te'_6[Sal]$. The system also updates $te'_3[Sal]$, since te'_3 and te'_6 were previously marked as duplicates.*

6. *Ben validates the repair Sam made.*

7. *The system should be able to capture that Bob is not that reliable when it comes to validating employee records, that Alice reports valid errors, and that Sam is reliable in repairing the salary data. Table $Employees\_v_2$ contains the fixed errors.*

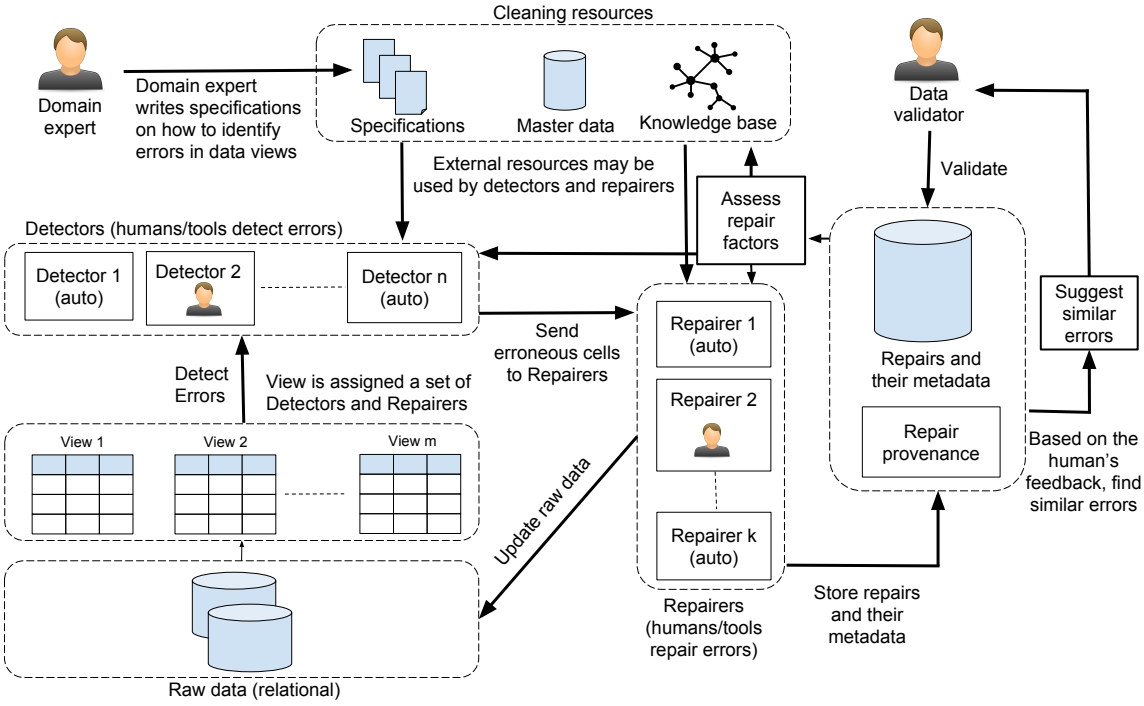
8. *We would like to enforce a functional dependency (FD) rule (ϕ_1: $Zip \rightarrow City$) on the Branches table. The cells marked in boldface violate ϕ_1. We use an FD repair tool, R_1, to automatically repair those violations. Table $Branches\_v_1$ is the repaired instance.*

Scenario 1 shows that it could be useful to have a holistic strategy to deal with various cleaning agents acting on different parts of the data. Such a framework has the potential to facilitate the human involvement in the cleaning pipeline at a high-level. However, realizing such a framework poses several challenges. Particularly, Scenario 1 raises several questions:

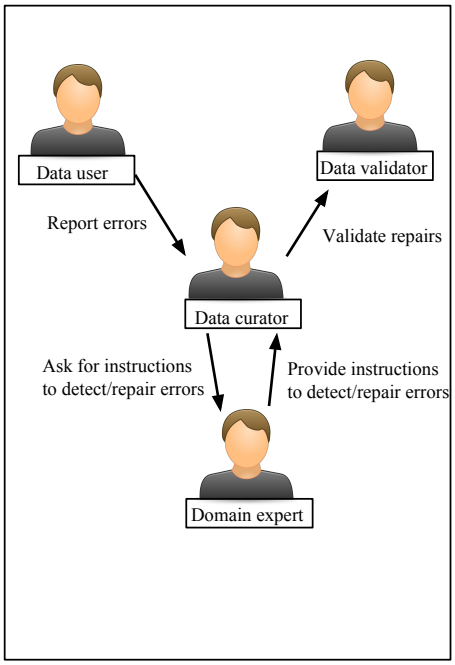
- *How do we know how confident Bob is about the validation he has performed on te_3? Asking another human to verify Bob's validation is expensive. How can we model Bob's knowledge on different parts of the data?*
- *How do we assess the quality of automatic tools, such as $Dedup$ and R_1 so that we know if a human validation is required after these tools are executed?*
- *How do we assess Alice's knowledge on the data?*
- *How can the system automatically route reported errors to humans with the right expertise to examine them?*
- *What if a certain repaired cell, say, $tb'_3[City]$ is deemed incorrect. Should we examine R_1 or the FD rule ϕ_1, or both? In other words, how can the system isolate the culprits in the data cleaning pipeline so that humans can easily debug cleaning decisions?*

The above questions are among many others that need to be addressed to effectively involve humans in the data cleaning pipeline. To this end, we propose a vision for an end-to-end data cleaning system that supports the following features:

- **Heterogeneity:** The system should be able to simultaneously support cleaning agents of different types, i.e., human, automatic or semi-automatic. Since different parts of the data can be cleaned by different agents, each agent receives part or all of the data as input.
- **Isolation:** The system should treat cleaning agents as black boxes while still enabling humans to detect, repair, and verify errors or bottlenecks (e.g., cleaning agents that are associated with wrong repairs) in the cleaning process. Thus, humans are isolated from the specific cleaning logic of a specific cleaning algorithm.
- **Human Cost Optimization:** The system should be able to reason about the expertise of different humans when assigning cleaning tasks. It should also account for the cost and expertise when involving a given human in a cleaning task.



(a) Architecture



(b) Example human interaction model

Figure 5.2.: Architecture (vision) and an example human interaction model

- **Accountability:** There are many factors involved in computing a given repair. Based on human feedback (e.g., human reports an error in a repaired cell), the system should automatically assess the reliability of different factors (e.g., agents, rules, etc.) that were involved in computing a repair over time. This assessment is crucial for humans to identify bottlenecks, i.e., factors associated with inaccurate repairs, in the data cleaning process.

5.2 Architecture Overview

5.2.1 Terminology

Consider a relational database D containing relations R_1, R_2, ..., R_n. Every relation R_i ($1 \leq i \leq n$) contains a set of attributes A_1^i, A_2^i, ..., A_k^i with domains $dom(A_1^i)$, $dom(A_2^i)$, ..., $dom(A_k^i)$ respectively. For the instance I_i of R_i containing tuples T, a cell c is the value of a tuple $t \in T$ in attribute $A \in R_i$, denoted $t[A]$.

Detector: Detectors are humans or programs that, given a set of cells as input, provide a set of cells that are potentially erroneous as output. Example detector programs are those that use data quality rules (e.g., Denial Constraints) to identify the cells that violate those rules.

Repairer: Repairers are humans or programs that update the input cells in a way that "fixes" the data errors.

Repair: We refer to an update to a set of cells C made by a Repairer R as a *repair*.

Accurate Repair: A repair is accurate if it contains cells with values that match the ground truth.

5.2.2 Architecture

Figure 5.2a illustrates the proposed architecture to implement our system vision. In a nutshell, there are four main components: Detectors, Repairers, Cleaning resources and Validators. All the Detectors and Repairers are treated as pluggable black boxes. One could

use any number of detection and repairing algorithms to clean the data. Since different agents can be involved to detect/repair different parts of the data, Detectors and Repairers are applied to different data views. Furthermore, Detectors and Repairers may use *cleaning resources* such as rules, masterdata, etc., to detect and/or repair the data. Cleaning resources are commonly produced by humans. We explore in Section 5.6 how the envisioned system should make it easy for humans to identify agents or cleaning resources that produce inaccurate repairs. Finally, in addition to detecting and fixing errors, humans are also able to validate the computed repairs, and based on their feedback, the system assesses the reliability of different factors that were involved in computing the repairs.

Data Cleaning job: The envisioned system allows humans to declaratively specify a data cleaning operation as a function of several parameters. Specifically, a data cleaning job is represented as the quadruplet $\langle C, D, R, V \rangle$ where: C is the set of input cells (cannot be empty), D is the set of Detectors to be used to detect errors in C, R is the set of Repairers to repair the errors found in C, V is the set of humans to validate the produced repairs. Using this representation, we can capture most of the cleaning scenarios. For example, if D and R are empty and V is not empty, then, the job will be a validation task on the cells in C.

Example 5.2.1 *In Example 5.1.1 (scenario 1), the cleaning jobs are represented as:*

$job_1 : \langle C = *,\ D = \emptyset,\ R = \emptyset,\ V = \{\text{“}Bob\text{”}\} \rangle$

$job_2 : \langle C = *,\ D = \{\text{“}Alice\text{”}\},\ R = *,\ V = * \rangle$

$job_3 : \langle C = \{tb[Zip] = *, tb[City] = *\},\ D = \{\phi_1\},\ R = \{R_1\},\ V = \emptyset \rangle$

job_1 *states that “Bob” can validate any cell in the data.*

job_2 *states that “Alice” can report errors in any data cell, and if she does, the system should automatically assign a repairer to update the data, and a validator to verify the update.*

job_3 *states that we are using* ϕ_1 *(in practice, there is a program that projects* ϕ_1 *on the data to extract violations, but for simplicity, we are only including the rule) to detect errors in all the Zip and City cells. The errors are then repaired using the FD-repair tool* R_1*. The repairs are not subjected to validation (*$V = \emptyset$*).*

5.3 Humans in the Cleaning Process

While several research efforts involve the human in specific cleaning problems (e,g,. Entity Resolution [78], Integrity Constraints [12], Data Fusion [79]), there is no proposal that involves humans for general data cleaning (regardless of the cleaning problem at hand). Furthermore, characterizing human expertise for the purpose of general data cleaning remains unexplored. Particularly, data cleaning efforts that use crowdsourcing [77, 78] assume that crowd workers are non-experts. On the other end of the spectrum, we have data cleaning methods [9, 12] that assume humans are experts whose feedback is assumed to be always correct. In practice, humans can have different degrees of expertise on different parts of the data. We shed some light to highlight key challenges that need to be addressed to realize this characterization.

5.3.1 Characterizing Human Expertise

Cleaning tasks: Humans interact in various ways in the cleaning process. Based on our vision, we list four human-driven tasks, referred to in this chapter by *cleaning tasks*, that a human-centric data cleaning system needs to support.

1. **Detection:** Humans should be able to report errors in a given set of cells.
2. **Repairing:** When errors are reported, humans should be able to fix those errors by updating the data to reflect accurate values.
3. **Validation:** Humans should be able to verify a repair that has been made by another cleaning agent (human or automatic).
4. **Specification:** Humans should be allowed to write specifications (e.g., FD rules) to detect data errors.

Human roles: The above interaction cases impose a distinction between different human roles in the cleaning process. Human roles in data cleaning are not well-studied. At a high level, we can think of a separation between human roles based on the knowledge of the

data, the domain, and the technical tools needed to update and transform the data. In the Detection task, the person reporting errors does not have to be technical. We refer to this person as the *Data User*. In the Repair task, because the human has to update the data with new values, the human has to possess the necessary technical background to perform the repair without introducing new data errors for technical reasons (e.g., a faulty SQL statement), we refer to this person as the *Data Curator*. In order to know *how* to repair a reported error, the *Data Curator* has to be knowledgeable enough about the erroneous data. In the Validation task, the humans validating the repairs have to be knowledgeable about the data they are asked to validate but do not have to be technical. For example, the system can ask a person yes/no questions about some data cells. We refer to this person as the *Data Validator*. In the Specification task, the person has to be a domain expert who can write specifications (for example, in the form of rules) that are then used to capture data errors. We refer to this person as the *Domain Expert*.

Data Expertise: Humans have different knowledge about different parts of the data. For example, a person working in the Sales department is probably more aware of the Sales data than someone working in the Marketing department. When assigning humans to cleaning tasks, it is important the system makes sure the assigned humans are knowledgeable enough in the data they are asked about. In a human-centric data cleaning system, every human has a history of the data cells they helped clean. Through the *Validation* task, the system can learn how *good* a given human is for a certain cleaning task and for a given cell. For example, a simple measure to quantify the expertise of a human h on data cells C, for a task T, is the following:

$$Expertise(h, C, T) = \frac{\#(h, C, T)}{\#validated(C, T)} \tag{5.1}$$

Equation 5.1 calculates the ratio of cells in which a human h performed a task T over to the number of cells in C that were subject to validation.

For example, we would like to measure the expertise of a human h in the detection task for a set of cells C. Assume h correctly reported errors in two of these cells.

The total number of cells that were validated by a Data Validator in C is 4. Therefore: $Expertise(h, C, T =$ "$Detection$") = $\frac{2}{4}$ = 0.5

Cost Model. Involving humans is generally expensive. It is important to be able to characterize the cost of involving a human to perform a certain cleaning task. For example, involving domain experts is generally more costly than involving ordinary data users. This cost should also take into consideration the availability of different human roles. For instance, if we have very few data curators, we would want to make sure they are assigned the most critical tasks only. Furthermore, it would be interesting to incorporate the cognitive effort of looking at the data to perform a cleaning task.

Human Budget. The human budget for a data cleaning job j could be expressed as a combination of many factors including the maximum number of humans available to perform a certain task, the total money cost to spend to perform a task, time limit, etc.

We are now ready to formally define a Human characterization of a human h.

Definition 5.3.1 *Human Characterization. A human h in a data cleaning scenario is represented as h: $\langle Role, Data, Cost, Expertise\rangle$ where $Role$ is the role of the human, $Data$ is the set of cells h is knowledgeable about, $Cost$ is the cost of involving h, $Expertise$ is a score that reflects how good h is for the role $Role$ in cells $Data$.*

5.4 Task Allocation

The envisioned system should allow users to define data cleaning jobs without explicitly stating the humans involved. Specifically, the system should be able to select from a pool of humans H with different characterizations, the right human for the right task. An example job is defined as follows:

$job_4 = \langle C = *, D = \{$"$Alice$"$\}, R = *, V = \emptyset\rangle$

In job_4, The set of repairers includes all the humans H available for this task. This makes the system responsible for assigning a repairer for the errors that $Alice$ detects. In our system, we are only interested in automatically assigning humans to cleaning tasks. Assigning automatic agents to cleaning tasks is outside the scope of the proposed vision.

Given a set of humans with their characterizations, the proposed system should be able to automatically assign cleaning tasks to them. We now discuss key building blocks that are needed to effectively assign cleaning tasks to humans.

5.4.1 Interaction Between Humans

It is crucial to develop an interaction model between different human roles to optimize the cleaning effort. Ideally, we should aim for an interaction model that produces the best cleaning results at the least human cost. In particular, a "good" interaction model should: (1) Minimize the communication overhead between different human roles; and (2) Account for all possible human-to-human interaction cases in the cleaning scenario. These cases are dictated by the set of human roles and their expertise. For instance, as illustrated in Figure 5.2b and using the roles we defined previously, the possible human-to-human interaction scenarios are the following:

- Data User reports errors to the Data Curator.
- Domain Expert provides specifications (e.g., rules, etc.) to the Data Curator to enforce on the data.
- Data Curator reports errors found in specifications to the Domain Expert.
- Data Validator validates fixes performed by the Data Curator.

5.4.2 Task Assignment

Given a data cleaning job j for cells C, a pool of humans, say H, and a budget, say B, the framework should assign automatically cells in C to humans in H (e.g., job_4 defined above). The assignment should guarantee the following properties: (1) **Coverage:** If the job is to be performed by humans only, every cell in C should be covered by at least one human; (2) **Maximize expertise**: The assigned humans should have good knowledge about cells in C; (3) **Minimize cost:** The human cost should not exceed Budget B.

Example 5.4.1 *Assume that we have a validation task on all the Sal cells of Table Employees in Figure 5.1 defined as the data cleaning job job_5 as follows:*

job_5 = $\langle C = Employees[Sal]$, $D = \emptyset$, $R = \emptyset$, $V = * \rangle$

Consider a pool of humans H = $\{Alice, Bob,$ and $Sam\}$, and a human budget ($B = 1$) for job_5 expressed (for simplicity) as the maximum number of humans involved in the task. $Alice$, Bob, and Sam have good knowledge on the following sets of cells $\{te'_1[Sal]$, $te_2[Sal]$,$te_3[Sal]$, $te_4[Sal]$, $te_5[Sal]\}$, $\{te_3[Sal]$, $te_4[Sal]\}$, and $\{te_5[Sal]\}$, respectively. In this scenario, the system should assign job_5 to $Alice$ only (since $B = 1$ and $Alice$ covers all the cells of Sal).

5.5 Cross-Agent Cost Optimization

Minimizing the human cost to repair the data has been the cornerstone of numerous research efforts [12]. However, when the human is not aware of the cleaning algorithm's logic, it becomes hard to achieve this goal. For instance, consider a Detector $Dedup$ that detects duplicate records using a clustering algorithm. $Dedup$ uses some similarity measure Sim to decide if a set of data points belong to the same cluster. Knowing how $Dedup$ works, if we want to validate its output, we could ask a human to verify if the closest points (using Sim) between the clusters are indeed not duplicates. In the case of our envisioned system, $Dedup$ would simply provide its output as clusters expressed in terms of records (duplicate records would share the same value of a designated attribute). In this case, involving the human usefully becomes more challenging.

We need to answer the following questions: (1) When we have human and automatic cleaning agents, what are the consequences of involving one over the other on human cost and data quality? (2) Given multiple humans that are assigned the same set of cells to repair, which human do we choose? (3) How do we schedule different cleaning jobs in order to achieve an optimal human cost and data quality?

5.5.1 Quantitative Cost Optimization

When we have humans and automatic agents that are assigned overlapping input data, which one should we prioritize? and what are the consequences for each choice? For example, in Example 5.1.1, what if Sam has also been assigned to repair cells $tb_2[City]$, $tb_3[City]$, $tb_2[Zip]$, $tb_3[Zip]$. In this case, the input to R_1 (automatic agent) overlaps with the input to Sam. This overlap is possible in practice. For example, one may want an automatic agent to clean a large amount of data while requiring the human to repair only a small subset of it. If we want to minimize human intervention, we can simply prioritize automatic agents over humans for a given set of cells. As a result, because Sam is assigned cells that are part of the input to R_1, we can simply save cost by not asking Sam to repair $tb_2[Zip]$, $tb_2[City]$, $tb_3[Zip]$, $tb_3[Zip]$, but we would still ask him to fix the Salary value in $te'_6[Sal]$ because this cell is not input to an automatic agent. While human intervention is minimized in this strategy, we note the following points:

- Human cost is minimized at the expense of data quality. That is, humans generally perform better repairs than automatic agents.
- This strategy can be suitable if the automatic agents provide high repair accuracy.

5.5.2 Qualitative Cost Optimization

This strategy gives preference to humans over automatic agents. As a result, the human cost will be higher compared to the previous strategy. In this strategy, when the input cells for an automatic agent overlap with those for a human agent, the system first invokes the automatic agent, and then asks the human to correct the overlapping cells. This way, the human updates will be ordered last and will not be undone by the automatic agent. Using this strategy, we note the following:

- Because humans are prioritized over automatic agents, it is expected that this strategy results in better data quality compared to the previous one.

- Human cost is high in this strategy. This strategy is suitable when invoking the automatic agents would result in a low repair quality.

5.6 Identification of Bottlenecks

There are several factors involved in repairing a cell, say c. We refer to these factors by $factors(c)$, where they include: (1) Detectors: Human or automatic agents that have flagged c's old value as erroneous; (2) Repairers: Agents (humans or automatic) that have computed the repair in c; (3) Cleaning resources: Resources used to compute c, which include Rules, Metadata, etc. (4) Data Validators: Humans that validated c as a correct repair (if c has been subject to validation). Therefore, the framework has to keep track of the provenance of every computed repair expressed in terms of all the factors that were involved to compute the repair for given cells.

After human validation, if a repair for c is deemed accurate (respectively, inaccurate), then every factor in $factor(c)$ should be rewarded (respectively, penalized). Providing this accountability will help identify factors that are commonly associated with inaccurate repairs. This assessment is crucial for humans as it helps them identify bottlenecks in the cleaning pipeline as a whole.

Scoring factors: One way to capture the reliability of different factors is to compute a score for each one of them that reflects how "good" each factor is. A simple way to capture the quality of a given factor $f \in factor(c)$ is the following:

$$Quality(f) = \frac{\#correct(f)}{\#validated(f)} \tag{5.2}$$

Equation 5.2 calculates the ratio of correct cells (as validated by a human) where f was involved over the total number of validated cells where f was involved.

Since they would result in inaccurate repairs, the "bad" factors would elicit more human feedback than the "good" ones. Therefore, identifying them is crucial to minimize the human cost in the cleaning process.

Scenario 2: Consider Example 5.1.1. We want to perform a new cleaning iteration with an additional FD rule that will be enforced on table $Branches\_v_1$ (Figure 5.1). Let us add an *incorrect* FD rule: $\phi_2 : City \rightarrow Zip$. This rule states that records that share the same City should have the same Zip code. This is in reality not correct because a city can have multiple zip codes. We now create a new data cleaning job: $job_6 : \langle C = \{tb'[Zip] = *, tb'[City] = *\}, D = \{\phi_1, \phi_2\}, R = \{R_1\}, V = \emptyset\rangle$

The set of violating cells will be $C^{\not\models}$ = $\{tb'_1[Zip],\ tb'_1[City],\ tb'_4[Zip],\ tb'_4[City],\ tb'_5[Zip], tb'_5[City]\}$. Let us assume that R_1 lifts the violation by setting $tb'_1[Zip] = 47904$. Let us call the repaired cell $tb''_1[Zip]$.

If we want to ask Jen, a human validator about the repairs computed by R_1, which violating cells should we ask her to validate? More importantly, how does the choice of cells we choose to validate affect the ability to isolate troublesome factors? Furthermore, how can we adjust the choice of cells to validate to our available human budget? To shed some light on answering those questions, we discuss the following key cases:

1. If we validate cells that were computed using many factors, we get an *aggregate* feedback on all the involved factors. For instance, asking Jen to validate $tb'_5[City]$ would provide a feedback about ϕ_1, ϕ_2 and R_1 (that cell was involved in two violations across different cleaning iterations). This is useful to get a feedback about many factors at once, however, it may not be good at isolating factors to identify the bottlenecks causing inaccurate repairs. This strategy is suitable when the cost of involving human validators is high. Therefore, this strategy allows us to have an idea about as many factors as possible using the least number of cells to hopefully identify a combination of factors that produced inaccurate repairs.

2. If we validate cells that were computed using few factors, we get a more fine-grained feedback about the involved factors. For example, asking Jen to validate $tb''_1[City]$ and $tb''_1[Zip]$ (which represent the repairs for cells $tb'_1[City]$ and $tb'_1[Zip]$ respectively) would isolate ϕ_2 as a problematic FD rule (since $tb'_1[Zip]$ and $tb'_1[City]$ vio-

lated ϕ_2 only). While this strategy provides a better isolation of factors, it involves more cells to be validated which translates into spending a higher human cost.

5.7 Related Work

There is a rich literature on Data Cleaning techniques and theory [7, 14, 80]. We discuss a few papers in two areas: general data cleaning systems and human-assisted data cleaning techniques.

General Data Cleaning Systems: A strongly related system to our proposal is the data cleaning system NADEEF [8]. Like our envisioned system, NADEEF adopts a system-approach to realize an end-to-end data cleaning framework that supports a number of data cleaning tasks (rule-based repairing, deduplication, etc.). NADEEF offers a programming interface so that users can implement detection and repairing components. As opposed to NADEEF, our framework not only supports automatic agents, but supports involving humans in all the stages of the data cleaning pipeline. Another related system is *KATARA* [77] which leverages the crowd and knowledge bases (KB) to clean dirty tables. *KATARA* is not rule-driven and can repair any cells in the input tuples (hence its categorization as a general data cleaning system). *KATARA* jointly uses the KB and the crowd feedback to identify correct and erroneous data in the input dirty table. Our vision is different from *KATARA* as we are considering humans with different expertise and roles (as opposed to using non-expert crowd workers). Furthermore, our envisioned system supports any number of cleaning agents for different cleaning tasks (integrity constraints, deduplication, etc.).

Human-Guided Data Cleaning: The idea of assisting humans in specific data cleaning problems (Entity Resolution, Schema transformations, etc.) has been widely studied [9, 12, 75, 76, 81, 82]. However, human involvement in these proposals is coupled to the underlying cleaning logic. Our proposal complements this line of work by enabling any number of cleaning tools to co-exist in the same platform while supporting multi-tool, algorithm-agnostic use cases that are otherwise not captured when using a tool to perform a specific data cleaning task.

The crowdsourcing literature [83] focuses mainly on scenarios that involve non-expert humans. However, in our framework, humans have multiple roles with varying degrees of expertise. Furthermore, our setup supports non-human agents as well.

5.8 Concluding Remarks

Scaling the generation and processing of big data often comes at the cost of its quality. We presented our vision for a framework that assists humans in all the major stages of the cleaning pipeline. We proposed several properties that need to be met to unlock the full potential of a given data cleaning scenario. We raised several questions that need to be addressed in order to bring such a vision to life. There are still many other questions that were not addressed in this vision such as data privacy, i.e., how can humans clean the data in the presence of privacy constraints? But we believe the proposed vision raises the central questions that we need to answer first to realize a human-centric data cleaning system.

6 CONCLUSIONS

In this dissertation, we addressed several data quality challenges in different settings. The process of data cleaning in the online setting does not come in a "one size fits all" fashion. We saw that we had to propose radically-different approaches to address different types of data errors, namely, duplicates and FD violations.

For online record linkage, we proposed a framework that enables query-time record linkage and fusion. We proposed a caching solution and developed a lookup strategy to match query answers to cache records efficiently. We then proposed mechanisms to iteratively update the cache with incoming query answers. We experimentally showed that our techniques provide a sharp gain in quality at a small time overhead. Furthermore, we enabled off-the-shelf record linkage and fusion tools to work in the online setting.

For online FD repairing, we first had to fundamentally rethink how FD repairing works. The traditional methods that enforce FDs on the data are quadratic as they require detecting all the pairs of tuples violating the FDs, and then lift the violations. We cannot afford to perform such a process at query-time, so we developed a novel way to approach FD repairing without the need to perform detection. In particular, we proposed a graph structure on top of which we developed repair algorithms that run in linear time. The key idea of our FD repairing framework is to project the FDs on the data and generate FD patterns. We devised different cases of interaction for FD patterns and developed a model to compose them and reason about their quality.

Furthermore, we adapted our FD repairing framework to work in the online setting. In particular, we implement a virtual integration system that integrates results from several Web sources at query time. We used this system as a test-bed to test our query-time FD repairing proposal. We showed that with small changes to our FD repairing framework, we were able to have a fully-functional query-time FD repairing system. In particular, we developed a method to compute the quality of FD patterns incrementally as query results

are processed in the system. The key takeaway from this project is that it is important to design cleaning algorithms that are setting-independent, and could be readily tuned to work in the offline, iterative or the online settings.

Last but not least, we discussed future directions in data cleaning. Since humans are the ultimate authority to validate data repairs, they have to be involved in the data cleaning pipeline in a smart way. Moreover, data cleaning is a process that typically involves two agents, namely, tools (e.g. FD repairing tool) and humans (e.g. domain expert). However, there is currently a big gap between these two agents. Typically, we either find tools that are fully automatic or those that involve the human in the cleaning logic of the tool. It is important to design data cleaning frameworks that can involve humans no matter what the underlying cleaning algorithm is.

REFERENCES

REFERENCES

[1] Experian. The data quality benchmark report: http://go.experian.com/global-research-report-2015, 2015.

[2] IBM Big Data and Analytics Hub. Extracting business value from the 4 v's of big data: https://goo.gl/dwd8hi.

[3] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[4] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *CoRR*, abs/1711.01299, 2017.

[5] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[6] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *Proceedings of VLDB Endowment*, 6(8):553–564, June 2013.

[7] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proceedings of VLDB Endowment*, 9(12):993–1004, 2016.

[8] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. NADEEF: A commodity data cleaning system. In *SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 541–552, New York, NY, USA, 2013. ACM.

[9] Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani, Jorge-Arnulfo Quiane-Ruiz, and Nan Tang. UGuide: User-guided discovery of FD-detectable errors. In *SIGMOD International Conference on Management of Data*, SIGMOD '17, pages 1385–1397, New York, NY, USA, 2017. ACM.

[10] Alexandra Meliou, Wolfgang Gatterbauer, Suman Nath, and Dan Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 505–516, New York, NY, USA, 2011. ACM.

[11] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *Transactions on Knowledge and Data Engineering*, 19(1), 2007.

[12] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *Proceedings of VLDB Endowment*, 4(5):279–289, February 2011.

[13] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1215–1230, New York, NY, USA, 2015. ACM.

[14] Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, October 2015.

[15] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. The LLUNATIC data-cleaning framework. *Proceedings of VLDB Endowment*, 6(9):625–636, July 2013.

[16] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *International Conference on Data Engineering (ICDE), Brisbane, Australia, April 8-12, 2013*, pages 458–469, 2013.

[17] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 53–62, New York, NY, USA, 2009. ACM.

[18] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 143–154, New York, NY, USA, 2005. ACM.

[19] El Kindi Rezig, Eduard Constantin Dragut, Mourad Ouzzani, Ahmed K. Elmagarmid, and Walid G. Aref. ORLF: A flexible framework for online record linkage and fusion. *International Conference on Data Engineering (ICDE)*, pages 1378–1381, 2016.

[20] El Kindi Rezig, Eduard C. Dragut, Mourad Ouzzani, and Ahmed Elmagarmid. Query-time record linkage and fusion over Web databases. In *International Conference on Data Engineering (ICDE)*, 2015.

[21] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. Incremental record linkage. In *Proceedings of VLDB Endowment*, 2014.

[22] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *International Conference on Data Engineering (ICDE)*, pages 244–255, March 2014.

[23] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh Basalamah. Tornado: A distributed spatio-textual stream processing system. *Proceedings of VLDB Endowment*, 8(12):2020–2023, August 2015.

[24] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, and Ahmed R. Mahmood. Pattern-driven data cleaning. *CoRR*, abs/1712.09437, 2017.

[25] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Computing Surveys*, 41:1:1–1:41, January 2009.

[26] Weiyi Meng and Clement Yu. *Advanced Metasearch Engine Technology*. Morgan & Claypool Publishers, 2010.

[27] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *SIGMOD International Conference on Management of Data*, pages 915–926, 2010.

[28] Pawel Jurczyk, James J. Lu, Li Xiong, Janet D. Cragan, and Adolfo Correa. FRIL: A tool for comparative record linkage. In *AMIA*, 2008.

[29] Peter Christen. Febrl - An open source data cleaning, deduplication and record linkage system with a graphical user interface. In *SIGKDD*, pages 1065–1068, 2008.

[30] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD International Conference on Management of Data*, 2009.

[31] Hung-sik Kim and Dongwon Lee. HARRA: Fast iterative hashed record linkage for large-scale data collections. In *EDBT*, pages 525–536, 2010.

[32] Nimrod Megiddo and Dharmendra S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer*, 37:58–65, 2004.

[33] Bo Zhao, Benjamin I. P. Rubinstein, Jim Gemmell, and Jiawei Han. A bayesian approach to discovering truth from conflicting sources for data integration. *Proceedings of VLDB Endowment*, 5(6), 2012.

[34] Flavio Chierichetti, Ravi Kumar, and Sergei Vassilvitskii. Similarity caching. In *PODS*, pages 127–136, 2009.

[35] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[36] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2005.

[37] Xuan Liu, Xin Luna Dong, Beng Chin Ooi, and Divesh Srivastava. Online data fusion. In *Proceedings of VLDB Endowment*, 2011.

[38] Shengli Wu and Sally McClean. Result merging methods in distributed information retrieval with overlapping databases. *Information Retrieval*, 2007.

[39] Yinglian Xie and David O'Hallaron. Locality in search engine queries and its implications for caching. In *IEEE International Conference on Computer Communications*, pages 1238–1247, 2002.

[40] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[41] Raymond A. K. Cox, James M. Felton, and Kee H. Chung. The concentration of commercial success in popular music: An analysis of the distribution of gold records. 19, 1995.

[42] Raaj Sah and Rajeev Kohli. Market shares: Some power law results and observations. *Harris School of Public Policy Studies, University of Chicago*, January 2004.

[43] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *PODS*, pages 233–242. ACM, 2006.

[44] Xin Luna Dong and Felix Naumann. Data fusion: Resolving data conflicts for integration. *Proceedings of VLDB Endowment.*, 2:1654–1655, 2009.

[45] Evangelos P. Markatos. On caching search engine query results. In *Computer Communications*, page 2001, 2000.

[46] Debabrata Dey, Vijay Mookerjee, and Dengpan Liu. Efficient techniques for online record linkage. *Transactions on Knowledge and Data Engineering*, 23, 2011.

[47] Thi Truong Avrahami, Lawrence Yau, Luo Si, and Jamie Callan. The FedLemur project: Federated search in the real world. *Journal of the Association for Information Science and Technology*, 2006.

[48] Jens Bleiholder, Samir Khuller, Felix Naumann, Louiqa Raschid, and Yao Wu. Query planning in the presence of overlapping sources. In *EDBT*, 2006.

[49] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of VLDB Endowment*, pages 918–929, 2006.

[50] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[51] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[52] Steven Euijong Whang, Marmaros David, and Hector Garcia-Molina. Pay-as-you-go entity resolution. *Transactions on Knowledge and Data Engineering*.

[53] Gianni Costa, Giuseppe Manco, and Riccardo Ortale. An incremental clustering scheme for data de-duplication. *Data Mining and Knowledge Discovery*, 2010.

[54] Michael J. Welch, Aamod Sane, and Chris Drome. Fast and accurate incremental entity resolution relative to an entity knowledge base. In *CIKM '12*.

[55] Ekaterini Ioannou, Wolfgang Nejdl, Claudia Niederée, and Yannis Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *Proceedings of VLDB Endowment*, 2010.

[56] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems*, 33(2):6:1–6:48, June 2008.

[57] Mohamed Yakout, Laure Berti-Équille, and Ahmed K. Elmagarmid. Don'T be SCAREd: Use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 553–564, New York, NY, USA, 2013. ACM.

[58] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proceedings of VLDB Endowment*, 10(11):1190–1201, August 2017.

[59] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. Sampling from repairs of conditional functional dependency violations. *Proceedings of VLDB Endowment*, 23(1):103–128, February 2014.

[60] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[61] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216, New York, NY, USA, 1993. ACM.

[62] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 114–121, Oct 1971.

[63] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. Messing up with BART: Error generation for evaluating data-cleaning algorithms. *Proceedings of VLDB Endowment*, 9(2):36–47, October 2015.

[64] Shuang Hao, Nan Tang, Guoliang Li, Jian He, Na Ta, and Jianhua Feng. A novel cost-based model for data repairing. *Transactions on Knowledge and Data Engineering*, 29(4):727–742, April 2017.

[65] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proceedings of VLDB Endowment*, pages 315–326, 2007.

[66] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of VLDB Endowment*, 1(1):376–390, August 2008.

[67] Xin Luna Dong, Barna Saha, and Divesh Srivastava. Less is more: Selecting sources wisely for integration. In *Proceedings of VLDB Endowment*, Proceedings of VLDB Endowment, pages 37–48. VLDB Endowment, 2013.

[68] Paolo Papotti, Felix Naumann, Sebastian Kruse, and El Kindi Rezig. Systems and methods for data integration. *Qatar Foundation*, US Patent US20160154830A1, 2013.

[69] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[70] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *Journal of Computer and System Sciences*, 73(4):610–635, June 2007.

[71] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, February 2005.

[72] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 378–393, London, UK, 2002. Springer-Verlag.

[73] Jef Wijsen. Database repairing using updates. *ACM Transactions on Database Systems*, 30(3):722–768, September 2005.

[74] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *International Conference on Data Engineering (ICDE)*, 2016.

[75] Sanjib Das, Paul Suganthan G. C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD International Conference on Management of Data*, 2017.

[76] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proceedings of VLDB Endowment*, pages 381–390, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[77] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD International Conference on Management of Data*, 2015.

[78] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. CrowdER: Crowdsourcing entity resolution. *Proceedings of VLDB Endowment*, 5(11):1483–1494, July 2012.

[79] Romila Pradhan, Siarhei Bykau, and Sunil Prabhakar. Staging user feedback toward rapid conflict resolution in data fusion. In *SIGMOD International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 603–618, 2017.

[80] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.

[81] Norases Vesdapunt, Kedar Bellare, and Nilesh Dalvi. Crowdsourcing algorithms for entity resolution. *Proceedings of VLDB Endowment*, 7(12):1071–1082, August 2014.

[82] A. Assadi, T. Milo, and S. Novgorodov. DANCE: Data cleaning with constraints and experts. In *International Conference on Data Engineering (ICDE)*, pages 1409–1410, April 2017.

[83] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *Transactions on Knowledge and Data Engineering*, 28(9):2296–2319, Sept 2016.