University of Modena and Reggio Emilia

"Enzo Ferrari" Engineering Department

International Doctorate in
Information and Communication Technologies (ICT)

XXXVI Cycle

Ph.D. Dissertation

Towards Data Integration On-Demand

Integrazione di dati on-demand

Candidate:
Luca Zecchini

Advisor:
Prof. Sonia Bergamaschi

Co-Advisors:
Prof. Giovanni Simonini
Prof. Felix Naumann

Director of the School:
Prof. Luigi Rovati

Abstract

Companies and organizations depend heavily on their data to make informed business decisions. Therefore, guaranteeing high data quality is critical to ensure the reliability of data analysis. Data integration, which aims to combine data acquired from several heterogeneous sources to provide users with a unified consistent view, plays a fundamental role to enhance the value of the data at hand. When data integration involves a limited number of sources, ETL (Extract, Transform, Load) is generally adopted as a paradigm: raw data is collected, cleaned, and stored in a data warehouse to perform analysis on it. Nowadays, big data integration needs to deal with millions of sources; thus, the paradigm is more and more moving towards ELT (Extract, Load, Transform). A huge amount of raw data is collected and directly stored (e.g., in a data lake), then different users can transform portions of it according to the task at hand. Hence, novel approaches to data integration need to be explored to address the challenges raised by this paradigm.

One of the fundamental building blocks for data integration is Entity Resolution (ER), which aims at detecting records that describe the same real-world entity, to consolidate them into a single consistent representation. ER is typically employed as an expensive offline cleaning step on the entire data before consuming it. Yet, determining which entities are useful once cleaned depends solely on the user's application, which may need only a fraction of them. For instance, when dealing with Web data, we would like to be able to filter the entities of interest gathered from multiple sources without cleaning the entire continuously growing data. Similarly, when querying data lakes, we want to transform data on-demand and return results in a timely manner. Hence, we propose BrewER, a solution to evaluate SQL SP queries on dirty data while progressively returning results as if they were issued on the cleaned data. BrewER tries to focus the cleaning effort on one entity at a time, according to the priority defined by the user through the ORDER BY clause. For a wide range of applications (e.g., data exploration), a significant amount of resources can therefore be saved.

Further, duplicates not only exist at record level, as in the case for ER, but also at dataset level. In the ELT scenario, it is common for data scientists to retrieve datasets from the enterprise's data lake, perform transformations for their analysis, then store back the new datasets into the data lake. Similarly, in Web contexts such as Wikipedia, a table can be duplicated at a given time, with the different copies having independent development, possibly leading to the insurgence of inconsistencies. Automatically detecting duplicate tables would allow to guarantee their consistency through data cleaning or change propagation, but also to eliminate redundancy to free up storage space or to save additional work for the editors. While dataset discovery research developed efficient tools to retrieve unionable or joinable tables, the problem of detecting duplicate tables has been mostly overlooked in the existing literature. To fill this gap, we therefore present SLOTH, a solution to efficiently determine the largest overlap (i.e., the largest common subtable) between two tables. The detection of the largest overlap allows to quantify the similarity between the two tables and spot their inconsistencies.

BREWER and SLOTH represent novel solutions to perform big data integration in the ELT scenario, fostering on-demand use of available resources and shifting this fundamental process towards a task-driven paradigm.

Keywords

Data integration · Entity resolution · Dataset discovery · ELT · Pay as you go

Sommario

Sempre più spesso le decisioni di aziende e organizzazioni sono basate sui dati di cui esse dispongono. Garantire la qualità di tali dati è fondamentale per poter effettuare analisi accurate e affidabili. L'integrazione dei dati consiste nel combinare dati acquisiti da molteplici sorgenti eterogenee per fornire all'utente finale una vista unitaria e coerente su tali dati. Si tratta perciò di un processo fondamentale per incrementare il valore dei dati disponibili. In passato, operando su numeri limitati di sorgenti, si è affermato il paradigma noto come ETL, che richiede di estrarre i dati grezzi, pulirli e immagazzinarli in un data warehouse per poterli poi analizzare. Al giorno d'oggi, operando su milioni di sorgenti, è invece sempre più diffuso il paradigma noto come ELT, per il quale una grande quantità di dati grezzi viene raccolta e immagazzinata senza trasformazioni, ad esempio in un data lake. Gli utenti possono poi pulire le porzioni di dati utili per le loro applicazioni. È pertanto necessario studiare soluzioni innovative per l'integrazione dei dati, maggiormente adatte alle nuove sfide che tale modello comporta.

Uno dei processi fondamentali per l'integrazione dei dati è la riconciliazione di entità, che consiste nell'individuare i profili che descrivono la stessa entità reale (duplicati) per consolidarli in un unico profilo coerente. Storicamente, questo processo viene effettuato sull'intero dataset prima di poterlo utilizzare, risultando spesso molto costoso. In molti casi, solo una porzione delle entità pulite si rivela poi utile per l'applicazione dell'utente finale. Ad esempio, operando su dati raccolti dal Web, è fondamentale poter filtrare le entità d'interesse senza dover pulire l'intera mole di dati in continua crescita. Allo stesso modo, quando si effettuano interrogazioni su un data lake, si vuole pulire solo la porzione di interesse, ottenendo i risultati nel minor tempo possibile. Per rispondere a tali esigenze abbiamo realizzato BrewER, una tecnica per eseguire interrogazioni SQL su dati sporchi emettendo progressivamente i risultati come se fossero stati ottenuti sui dati puliti. BrewER focalizza il processo di pulizia su un'entità alla volta, in base a una priorità definita dall'utente nella clausola ORDER BY. Per molte applicazioni, come l'esplorazione dei dati, BrewER consente di risparmiare una grande quantità di tempo e risorse.

I duplicati non esistono solo a livello di singoli profili, ma anche a livello di dataset. È infatti comune ad esempio che un data scientist per le proprie analisi effettui trasformazioni su un dataset presente nel data lake aziendale, immagazzinando poi anche la nuova versione ottenuta all'interno del data lake stesso. Situazioni simili si verificano nel Web, ad esempio su Wikipedia, dove le tabelle vengono spesso duplicate e le copie ottenute hanno uno sviluppo indipendente, con la possibile insorgenza di inconsistenze. Individuare automaticamente queste tabelle duplicate consente di renderle coerenti con operazione di pulizia dei dati o propagazione delle modifiche, oppure di rimuovere le copie ridondanti per liberare spazio di archiviazione o risparmiare futuro lavoro agli editori. La ricerca di tabelle duplicate è stata perlopiù ignorata dalla letteratura esistente. Per colmare questa mancanza abbiamo quindi realizzato SLOTH, una tecnica che, date due tabelle, consente di determinarne la più grande sottotabella in comune, consentendo di quantificarne la similarità e di rilevare le possibili inconsistenze.

BREWER e SLOTH rappresentano soluzioni innovative per l'integrazione dei dati nello scenario ELT, utilizzando le risorse a disposizione su richiesta e indirizzando il processo di integrazione dei dati verso un approccio orientato alle applicazioni.

Parole chiave

Integrazione dati · Deduplicazione dati · Tabelle correlate · ELT · Pay as you go

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

Data is everywhere: available for searching on the Web, made accessible by the public administration as *open data*[1], collected using sensors and IoT devices. In practice, most aspects of our lives are transformed into data, and consequently in information with economic value, in a process known as *datafication* [Cukier and Mayer-Schoenberger, 2013]. Being able to analyze data therefore plays a fundamental role in many vital sectors of our society, from business [Popovič et al., 2018] and healthcare [Belle et al., 2015] to public administration [Kim et al., 2014] and smart cities [Bachechi et al., 2022]. More and more companies and organizations rely on the analysis of their data to take informed business decisions, a practice commonly referred to as *data-driven decision making* [Brynjolfsson and McElheran, 2016]. In addition to traditional techniques, Artificial Intelligence (AI) is widely (and increasingly) used in data analysis, with several models, especially solutions based on deep learning [LeCun et al., 2015], requiring large amounts of data for their training and testing, often in a labeled form.

Nevertheless, in many cases data scientists and practitioners have to work with data presenting quality issues [W. Fan and Geerts, 2012]. For instance, it may contain wrong or outdated values, some information may be missing, datasets may contain duplicates, and some annotations and labels may be incorrect or inconsistent (e.g., due to the presence of multiple annotators adopting different policies). Data quality represents a serious concern [Batini et al., 2009; Ehrlinger and Wöß, 2022], since data analysis can produce correct and meaningful results only if it is performed on input data of good quality, while input data with quality issues may significantly affect its outcome (i.e., *garbage in, garbage out*) and therefore jeopardize the goodness of final business decisions. Companies and organizations may suffer signifi-

[1]`https://opendefinition.org/`

cant undesired additional costs due to unreliable analysis results [Haug et al., 2011]. In fact, even the best AI models may perform badly on poor quality data [Budach et al., 2022], while on the other hand ensuring the quality of the data at hand can significantly improve the results of the analysis keeping the model unchanged. Such considerations, apparently obvious but often overlooked in practice [Sambasivan et al., 2021], led to an increasing demand for a *data-centric* approach to AI [Jarrahi et al., 2023], putting emphasis on the quality of the data rather than further improvements of state-of-art models.

To ensure the quality of the dataset at hand, users are required to perform a process of *data preparation and cleaning* [Fernandes et al., 2023], covering a plethora of different operators, also known as *preparators* [Hameed and Naumann, 2020], to fix possible issues present in the data. For instance, the practitioner might be required to locate missing values and outliers, check the presence of type-mismatched data, split or merge columns, etc. In particular, note that even if data preparation and data cleaning are often used as synonyms (as also done in this thesis), the latter mostly denotes corrections performed on the data at a semantic level (e.g., data deduplication or missing value imputation), while the former covers syntactic transformations [Hameed and Naumann, 2020]. Moreover, to maximize the information value of the dataset at hand, a practitioner is often required to enrich it with information acquired from further related sources, hence performing *data integration* [X. L. Dong and Srivastava, 2015]. For instance, further datasets might provide additional entities to integrate into the dataset at hand or additional attributes describing further aspects of the entities in the dataset, but also different representations of such entities that allow to assess the correctness of the information contained in the dataset.

Although it is essential, guaranteeing data quality is often not straightforward, and may require high costs in terms of time and resources. A famous survey [Press, 2016] indeed estimates that data scientists spend around 80% of their time to prepare their data (60% for cleaning and organize the data at hand, but also 19% for collecting datasets, and 3% for building training sets), while only the remaining part is dedicated to proper data science tasks, such as mining data for patterns (9%) or refining algorithms (4%). Moreover, a similar percentage of data scientists considers these steps as the least enjoyable part of their work. In fact, data preparation is a trial-and-error process that typically involves countless iterations over the data to define the best pipeline of operators for a given task. Further, different operators do not have the same impact on the downstream models [P. Li et al., 2021; Abdelaal et al., 2023], and some aspects of this process include a subjective component given by the decision criteria adopted by the different practitioners (a common phenomenon that affects data labeling or annotation). Similarly, detecting useful datasets for the task at hand may require a significant effort too, especially inside large corpora, where scalability becomes a crucial challenge [Paton et al., 2023].

Given the importance of data quality and the challenges that it requires to overcome, scientific research has made great efforts to support data scientists and practitioners in such tasks. Nevertheless, many issues remain open and need to be solved. In particular, this thesis mostly focuses on the topic of data integration, which constitutes the main research area of my PhD.

1.1 Data Integration

Data integration [X. L. Dong and Srivastava, 2015] is the process that combines data acquired from multiple autonomous sources to provide users with a unified consistent view on this data. Hence, data integration plays a fundamental role to enhance the value of the data at hand, allowing to combine it with relevant information available in other data sources. The data integration process is composed of three major tasks: *schema alignment*, *entity resolution*, and *data fusion*.

To provide a high-level intuition about the data integration process and its three tasks, let us consider the toy example illustrated in Figures 1.1 and 1.2. Figure 1.1a depicts an excerpt from the CD catalog of a music shop. For each album, the catalog stores a unique identifier (`ID`), its title (`Title`), the artist that recorded it (`Artist`), and the number of copies available in the store, ready to be sold (`Items`). The covers of the albums, reported on the left of each row, show the real-world object that the row is describing. Since the information collected about the albums is very limited, the employee in charge of managing the catalog would like to enrich it. To avoid the manual insertion of the values for additional attributes, he decides therefore to integrate it with the data contained in a Web table about rock albums, an excerpt of which is illustrated in Figure 1.1b. Compared to the CD catalog, beyond the title of the album (`Album`) and the name of the artist (`Artist`), the table contains information about the year of release (`Year`), the duration of the album expressed in minutes (`Length`), and the record label responsible of the release (`Label`).

Schema alignment [Rahm and Bernstein, 2001] has the goal of correctly aligning the schemas of tables from different sources. In practice, schema alignment has to solve the semantic ambiguity of the attributes. In fact, it is common that a conceptual information is modeled differently across multiple sources. For instance, the name of a person can be modeled using

	ID	Title	Artist	Items
	1	Fire of unknown origin	BÖC	3
	2	Ziggy Stardust	D. Bowie	18
	3	Rumours	Fleetwood Mac	15
	4	30:30	Pogues	5
	5	Monster	REM	8
	6	Nebraska	B. Springsteen	12

(a) An excerpt from the CD catalog of a music shop.

	Album	Artist	Year	Length	Label
	Atom Heart Mother	Pink Floyd	1970	52:06	Capitol
	Fire of Unknown Origin	Blue Öyster Cult	1981	39:06	Columbia
	Fleetwood Mac	Fleetwood Mac	1975	42:12	Reprise
	Monster	R.E.M.	1994	49:15	Warner Bros.
	Nebraska	Bruce Springsteen	1982	40:50	Columbia
	Rubber Soul	The Beatles	1965	34:59	Parlophone
	Rumours	Fleetwood Mac	1977	38:55	Warner Bros.
	The Rise and Fall of Ziggy Stardust and the Spiders from Mars	David Bowie	1972	38:29	RCA
	Tubular Bells	Mike Oldfield	1973	49:18	Virgin

(b) An excerpt from a Web table about rock albums.

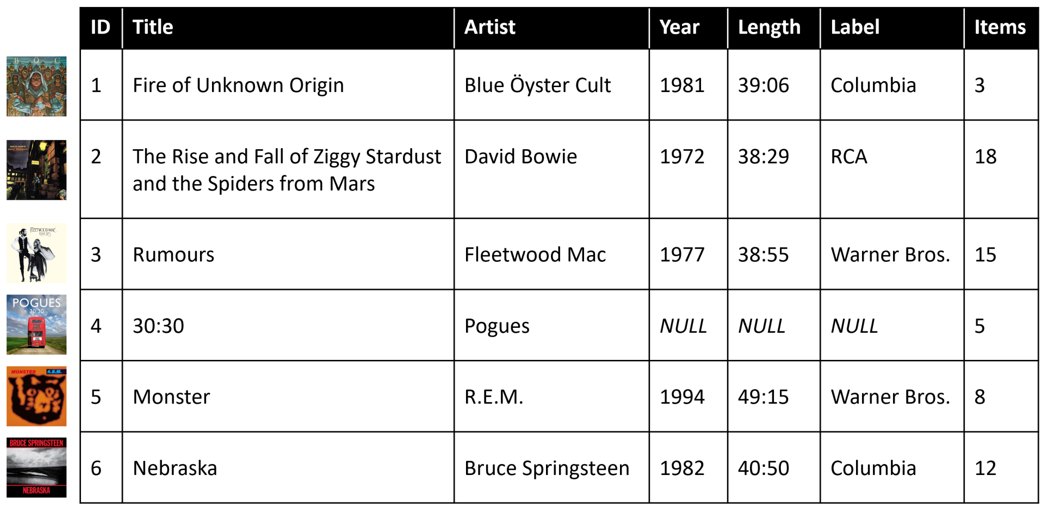
Figure 1.1: Excerpts from two datasets to be integrated.

a single attribute `Name`, but also using the pair of attributes `FirstName` and `LastName`, or alternatively `Name` and `Surname`. In the toy example, while the columns about the artist share the same header in both tables, a different naming is used for the album title, denoted as `Title` in the CD catalog (Figure 1.1a) and as `Album` in the Web table (Figure 1.1b).

Schema alignment typically operates in three consecutive steps: *(i)* the definition of the *mediated schema*, designed to capture the main aspects of the domain, on which the users perform their queries; *(ii) attribute matching*, that associates the attributes from the schema of each source to their representation in the mediated schema; *(iii) schema mapping*, exploiting the correspondences defined by the attribute matching to reformulate every user query (performed on the mediated schema) into a set of specific queries, one for each local source.

Entity resolution [Papadakis et al., 2021a] is the task of detecting the instances (i.e., *records* or, more generally, *profiles*) that describe the same real-world object (i.e., *entity*). Records describing the same entity are denoted as *matches*. In this case, the challenge is to solve the ambiguity in the entity representations. In fact, the same entity can be described in multiple ways, and linking such different representations to the referred entity might not be obvious. In the toy example, information about each album was inserted manually over time into the CD catalog (Figure 1.1a) by multiple employees of the store, who often used abbreviations or different conventions to represent the artists and the album titles. In the `Artist` column, acronyms were often used for some band names composed of more than two words (e.g., *BÖC* for *Blue Öyster Cult*), and only the initial letter was sometimes reported for the first names of some artists (e.g., *D. Bowie* for *David Bowie* or *B. Springsteen* for *Bruce Springsteen*). Similarly, the title of the album *The Rise and Fall of Ziggy Stardust and the Spider from Mars* was shortened as *Ziggy Stardust* and *30:30 - The Essential Collection* simply denoted using the main title. Finally, the bands *R.E.M.* and *The Pogues* are sometimes represented in the catalog as *REM* and *Pogues*, respectively.

Also entity resolution is generally performed with three major steps [Christen, 2012b]: *(i) blocking*, which groups records in different *blocks* according to some similarity criteria, discarding obvious non-matches (mitigating the inherently quadratic complexity of the entity resolution problem, which would require to compare all possible pairs of records); *(ii) entity matching*, which compares each pair of records co-occurring in the same block to determine if it constitutes a match or not; *(iii) entity clustering*, which builds on the detected pairwise matches to determine consistent clusters of records that refer to the same entity. The task of entity resolution, which represents one of my main research areas, is described



ID	Title	Artist	Year	Length	Label	Items
1	Fire of Unknown Origin	Blue Öyster Cult	1981	39:06	Columbia	3
2	The Rise and Fall of Ziggy Stardust and the Spiders from Mars	David Bowie	1972	38:29	RCA	18
3	Rumours	Fleetwood Mac	1977	38:55	Warner Bros.	15
4	30:30	Pogues	*NULL*	*NULL*	*NULL*	5
5	Monster	R.E.M.	1994	49:15	Warner Bros.	8
6	Nebraska	Bruce Springsteen	1982	40:50	Columbia	12

Figure 1.2: An excerpt from the CD catalog after data integration.

more extensively at the beginning of Chapter 2, in Section 2.1.

After detecting the cluster of records that refer to the same entity, data fusion [Bleiholder and Naumann, 2008] consolidates them into a single representative record for that entity with consistent values for its attributes. The choice of the value for each attribute in the representative record can follow different criteria. For instance, a source can be considered as more reliable, hence preferred over the others. Alternatively, it is possible to choose the most frequent value carried for that attribute by the records inside the cluster, or using aggregation functions for numerical attributes to maintain for instance the maximum value, the minimum value, or the average of the values. In the toy example, the Web table in Figure 1.1b is defined as the source to trust in case of inconsistencies with the CD catalog (Figure 1.1a) on the two common attributes (i.e., the artist name and the album title), obtaining the result depicted in Figure 1.2. Note that the album *30:30* is not present in the Web table about rock albums (Figure 1.1b), since it was categorized by the website as a folk album. Thus, the cells of the new columns (i.e., `Year`, `Length`, and `Label`) present missing values in its row.

Finally, *dataset discovery* [Paton et al., 2023], i.e., the retrieval of further datasets related to the dataset at hand according to specified relevance criteria, plays a fundamental role for performing data integration in many real-world scenarios. In fact, while in some cases (as in the previous toy example) the datasets to integrate are already defined, in many other cases the practitioner has a dataset and wants to retrieve further datasets from a corpus that may add valuable information to it. In this case, it is fundamental to have efficient solutions to retrieve related datasets in the corpus in an automated way, according to the relevance criteria (i.e., the type of relatedness) defined by the practitioner (e.g., joinability, unionability, or duplicate detection). The task of dataset discovery, touched by the contributions presented in this thesis, is described in more detail at the beginning of Chapter 3, in Section 3.1.

Note that similar concepts to data integration are *data harmonization* and *data federation*. The idea of data harmonization is also strictly connected to data preparation and cleaning, with the goal of unifying data from different sources into a composite dataset with a consistent, standardized, and comprehensive format[2], for instance to perform analysis on it. This term, which is becoming popular in industry, is frequently used in different research areas, such as biology and medicine [Doiron et al., 2013; Rolland et al., 2015; Mirzaalian et al., 2016]. Data federation can be viewed as a *virtual* approach to data integration instead, enabling users to query and aggregate data from multiple sources without moving it from its original source[3], as supported for instance by systems such as Denodo[4] [Gu et al., 2022].

1.2 From ETL to ELT

Due to its fundamental role, data integration represents one of the longstanding challenges in data management. Therefore, according to the evolution of technology, the scenario in which data integration needs to be performed has changed a lot over time [X. L. Dong and Srivastava, 2015]. In fact, data integration was historically needed to operate on relatively limited quantities of data sources (often relational databases), for instance to integrate *data silos* autonomously managed by distinct departments of the same company [Bergamaschi et al., 1999]. Nowadays, data integration needs to work efficiently in an extremely different scenario, which often requires to take into account millions of heterogeneous sources, such as databases, Web tables, open data, and data collected from sensors or IoT devices [Bergamaschi et al., 2018].

When data integration involved a limited number of sources, *ETL (Extract, Transform, Load)* emerged as the most popular paradigm [Vassiliadis et al., 2002]. As suggested by the acronym, ETL is composed of three main steps. First, data is extracted from multiple sources, potentially heterogeneous. Then, this data is transformed into the desired format, performing

[2]`https://www.tibco.com/reference-center/what-is-data-harmonization`
[3]`https://www.tibco.com/reference-center/what-is-a-data-federation`
[4]`https://www.denodo.com/`

data preparation and cleaning operations. Finally, the transformed data is loaded into a *data warehouse* [Inmon, 1992; Chaudhuri and Dayal, 1997], enabling data scientists to perform business intelligence operations on the integrated data in its cleaned version.

Subsequently, the explosion of *big data* radically changed the scenario. Nowadays, big data integration often needs to deal with a huge amount of data scattered across millions of sources, extremely heterogeneous in their types and formats. Data stored in relational or NoSQL database management systems, data collected from the Web, especially Web tables [Cafarella et al., 2008; Bleifuß et al., 2021b], open data made available by public administrations [Miller, 2018], datasets purchased from data marketplaces [Azcoitia and Laoutaris, 2022], data acquired from sensors and IoT devices [Marjani et al., 2017]: each of these types of data has its own peculiar features and may therefore require dedicated procedures for its management. Further, even sources of the same type may differ significantly, for instance by adopting very different formats for storing their data, following various encodings and conventions (e.g., for representing dates or numeric values), etc.

In addition to the presented heterogeneity issues, many sources are also extremely dynamic [Tatbul, 2010]: new data is generated very frequently (as in the case of sensors or stock market) and possibly even existing data is updated very often (for instance, in Web pages following ongoing events). Moreover, data sources may present different levels of data quality and accuracy [Berti-Équille and Borge-Holthoefer, 2015]. All these problems, concisely described by the notorious 4 Vs of big data [X. L. Dong and Srivastava, 2015], i.e., *volume*, *variety*, *velocity*, and *veracity*, make data integration in such a context very challenging. The need to overcome these issues led researchers and practitioners to perform significant changes to established data integration paradigms and techniques, making them more suitable to the new scenario.

In particular, since applying the ETL approach to the case of big data would be prohibitively expensive and often technically unfeasible, the paradigm is more and more moving from ETL towards *ELT (Extract, Load, Transform)* [Furche et al., 2016]. Differently from ETL, in the ELT paradigm a huge amount of raw data is collected and directly stored as it is, for instance in a data lake [Nargesian et al., 2019; Nargesian et al., 2020; Armbrust et al., 2020]. Then, according to the task at hand, practitioners can transform and integrate useful portions of this large data corpus, for instance to perform business intelligence operation on the obtained clean data or to store it in a relational database management system to efficiently run queries on it [Gagliardelli et al., 2022b; Gagliardelli et al., 2023].

1.3 Data Integration On-Demand

Data lakes and large table corpora pose many challenges to the different tasks of data integration, which are therefore required to significantly change established paradigms and approaches to meet the needs raised by this new scenario. Indeed, in many situations nowadays it is not possible (or anyway not efficient) to operate on the entire amount of data with an expensive and time-consuming *offline* approach. This is for example the case of dynamic sources, where data changes very frequently and gets outdated quickly, hence timely results are needed. Further, this also happens anytime the practitioner is only interested in a certain portion of the data (e.g., when running queries to explore a novel dataset), hence cleaning the entire data would be a waste of time and resources. Therefore, novel solutions are usually required to handle only the data effectively needed for the task at hand, returning results in a timely manner, often in a *pay-as-you-go* fashion. This allows to save time, resources, and money (e.g., in case of pay-as-you-go contracts, popular in the cloud), producing results in a limited amount of time so that they can be useful to the practitioner.

The case of entity resolution clearly exemplifies this situation. In fact, entity resolution is typically employed as an expensive offline cleaning step on the entire data before consuming it. Yet, determining which entities are useful once cleaned depends solely on the user's application, which may need only a fraction of them. For instance, when dealing with Web data, we would like to be able to filter the entities of interest gathered from multiple sources without entirely cleaning the continuously growing datasets. Similarly, when querying data lakes, we want to transform only the portion of data interesting for the query and return results in a timely manner.

To overcome the limitations of existing approaches, researchers directed their efforts in two distinct directions. On one hand, *query-driven* approaches [Altwaijry et al., 2013; Altwaijry et al., 2015] aim to perform entity resolution only on the portion of data which might be useful to answer the issued query. On the other hand, *progressive* approaches [Whang et al., 2013; Papenbrock et al., 2015a; Firmani et al., 2016; Simonini et al., 2018] prioritize the comparisons that most likely lead to the detection of a match, maximizing the number of matches detected in a certain amount of time.

The approach presented in this thesis, called *entity resolution on-demand* and implemented by BrewER [Simonini et al., 2022; Zecchini et al., 2023; Simonini et al., 2023], conciliates and goes beyond the two described research directions. BrewER evaluates SQL SP queries on dirty data while progressively returning results as if they were issued on the cleaned version of this data, allowing therefore the user to run *clean queries on dirty data*.

BrewER, described extensively in Chapter 2, tries to focus the cleaning effort on one entity at a time, according to the priority defined by the user through the ORDER BY clause. Hence, for a wide range of applications (e.g., data exploration), it can save a significant amount of resources.

Another task that has been significantly affected by the new scenario is dataset discovery. In fact, having to deal with very large corpora of tabular datasets (e.g., Web tables, tables stored in data lakes, etc.), disposing of solutions to efficiently detect in an automated way datasets in such corpora related to the one at hand achieves a paramount importance. Hence, several efficient methods have been proposed for the detection of *related tables* [Das Sarma et al., 2012] in large table corpora, especially *joinable tables* [E. Zhu et al., 2016; E. Zhu et al., 2019; Esmailoghli et al., 2022] and *unionable tables* [Nargesian et al., 2018].

Nevertheless, duplicates not only exist at record level, as in the case for entity resolution, but also at dataset level. In the ELT scenario, it is common for data scientists to retrieve datasets from the enterprise's data lake, perform transformations for their analysis, then store back the new datasets into the data lake. Similarly, in Web contexts such as Wikipedia, a table can be duplicated at a given time, with the different copies having an independent development, possibly leading to the insurgence of inconsistencies. Automatically detecting duplicate tables would allow to guarantee their consistency through data cleaning or change propagation [Bleifuß et al., 2018], but also to eliminate redundancy to free up storage space or to save additional work for the editors. However, the problem of detecting duplicate tables has been mostly overlooked in the existing literature.

Sloth [Zecchini et al., 2024], the second novel solution presented in this thesis (where it is described extensively in Chapter 3), aims to fill this gap by efficiently determining the *largest overlap* (i.e., the largest common subtable) between two tables. The detection of the largest overlap allows to quantify the similarity between the two tables and spot their inconsistencies. Further, Sloth can be useful to multiple additional tasks, such as the detection of potential copying across different sources or the automatic detection of candidate multi-column joins, which is an open challenge in literature.

1.4 Contributions

The key contributions of this thesis are two novel solutions to perform big data integration in the ELT scenario, fostering on-demand use of available resources: BrewER and Sloth.

BrewER is a novel solution to perform entity resolution on-demand, allowing users to run SQL SP queries directly on dirty data obtaining the progressive emission of clean entities as if they were issued on the cleaned version of this data, according to a priority defined by the user. BrewER is extensively described in Chapter 2, which also contains a more detailed overview of entity resolution (Section 2.1), from the historical offline paradigm to the novel on-demand approach.

Sloth is a novel solution to efficiently determine the largest overlap (i.e., the largest common subtable) between two tables, allowing users to detect different and possibly inconsistent versions of the same table, then ensure their consistency through data cleaning and change propagation or delete redundant copies. Sloth is extensively described in Chapter 3, which also provides a description of the state-of-the-art solutions for dataset discovery (Section 3.1) to better understand what are the related approaches and why Sloth fills a relevant gap in the existing literature.

Finally, Chapter 4 concludes the thesis with an overview of the main contributions presented in the previous chapters and some research directions for further developing the novel solutions described in the thesis.

Chapter 2

BrewER

This chapter provides the detailed description of BrewER, a novel solution designed to perform *entity resolution on-demand*, allowing users to run SQL SP queries on dirty data obtaining the progressive emission (according to the specified priority) of the clean results as if they were issued on the cleaned version of the same data. BrewER was presented in a dedicated research paper at VLDB 2022 [Simonini et al., 2022] and its related demonstration at VLDB 2023 [Zecchini et al., 2023], also appearing in a restricted version as a discussion paper at SEBD 2023 [Simonini et al., 2023]. The code for BrewER is openly available on GitHub[1].

This chapter is structured as follows. First, Section 2.1 provides deeper insight into entity resolution, describing its main challenges and the different steps that compose the entity resolution pipeline, as it is designed in the traditional *batch* approach. Section 2.2 shows the shortcomings of this approach in some frequent real-world scenarios, introducing BrewER as the proposed solution. BrewER is then compared to related approaches in Section 2.3, highlighting their differences and the advantages of this novel solution. Section 2.4 introduces some preliminary definitions and the formalization adopted in the description of the proposed algorithm, illustrated in detail in Section 2.5.2. Section 2.6 depicts some example real-world use cases that can significantly benefit from BrewER, while Section 2.7 concludes the chapter by reporting the details of its experimental evaluation.

2.1 Entity Resolution

Entity Resolution (ER) [Christophides et al., 2021] aims at detecting *records* (or, more generally, *profiles*) in a dataset or across multiple datasets that de-

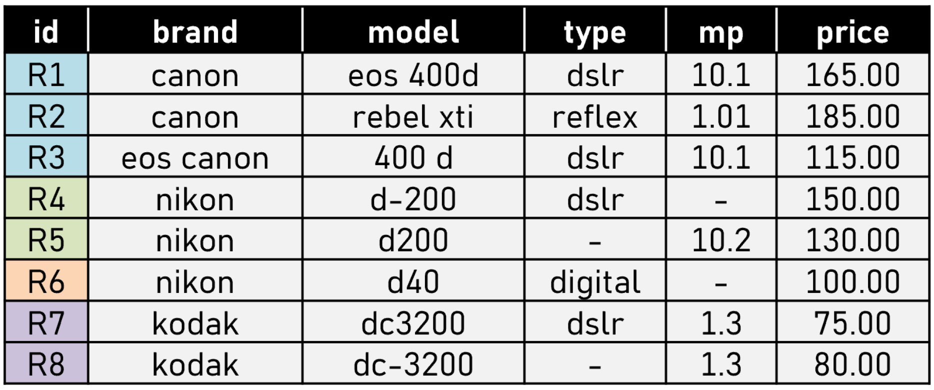
[1]`https://github.com/dbmodena/BrewER`

scribe (or refer to) the same real-world object (i.e., *entity*). Entity resolution is also known under multiple alternative names with slightly different shades of meaning, such as *record linkage* [Christen, 2012b], *duplicate detection* [Elmagarmid et al., 2007; Naumann and Herschel, 2010], *deduplication* [Chu et al., 2016], or sometimes, using a synecdoche, *entity matching*. Entity resolution can be performed on a single dataset containing duplicates to obtain its cleaned version (this process is denoted as *dirty entity resolution* or deduplication) or to link the different representations of the same entity across multiple clean datasets (this process is denoted instead as *clean-clean entity resolution* or record linkage), as in the toy example about the CD catalog described in Chapter 1.

As previously stated, entity resolution plays a fundamental role in data integration and, more generally, to ensure the quality of the data at hand, with a notable impact on several real-world scenarios. For instance, entity resolution can be used to link data about the same patient across multiple health databases, data about the same product across multiple catalogs, data about the same route from multiple travel company websites, etc. Entity resolution is also used for *master data management* in enterprise information systems [Loshin, 2008], to detect for instance multiple representations of the same customer or product.

Entity resolution is therefore a longstanding problem in data management. In fact, the *father* of entity resolution can be identified as the medical doctor and statistician Halbert L. Dunn, who firstly defined it in the context of vital statistics (especially death clearances) in 1946 [Dunn, 1946], before the formalization carried out by Ivan P. Fellegi and Alan B. Sunter in 1969 [Fellegi and Sunter, 1969].

The first challenge that entity resolution needs to overcome is the heterogeneity among the different representations of the same entity, which can sometimes make it very hard to correctly link a record to the described entity. Let us consider for instance the example reported in Figure 2.1, representing a dirty dataset containing records about cameras scraped from several e-commerce websites (e.g., Amazon, eBay, Alibaba, etc.). For each record, the dataset contains the unique identifier (`id`), the brand (`brand`), the model name (`model`), the type of camera (`type`), the resolution in megapixels (`mp`), and finally the price (`price`). The dataset contains many duplicate records that describe the same entity. This is denoted in Figure 2.1 through the color of the record ID cell, representing the following clusters of matching records: {R1, R2, R3}, {R4, R5}, {R6}, {R7, R8}. Records in the same cluster can differ significantly, due for instance to the adoption of different naming conventions (e.g., *d-200* vs. *d200*), the presence of typos (e.g., 1.01 instead of 10.1 MP), and in general of missing or inconsistent values, includ-



id	brand	model	type	mp	price
R1	canon	eos 400d	dslr	10.1	165.00
R2	canon	rebel xti	reflex	1.01	185.00
R3	eos canon	400 d	dslr	10.1	115.00
R4	nikon	d-200	dslr	-	150.00
R5	nikon	d200	-	10.2	130.00
R6	nikon	d40	digital	-	100.00
R7	kodak	dc3200	dslr	1.3	75.00
R8	kodak	dc-3200	-	1.3	80.00

Figure 2.1: A dataset of cameras scraped from multiple e-commerce websites. The color of the record ID cell denotes the described entity.

ing cases of homonymy and synonymy that might be extremely difficult to detect without some specific domain knowledge. For example, the Canon EOS 400D camera model (represented by records R1, R2, and R3) is sold in North America as Digital Rebel XTi and in Japan as EOS Kiss Digital X.

Further, a second relevant challenge is related to the inherently quadratic complexity of the entity resolution problem. In principle, it would indeed require to compare all possible pairs of records in the dataset to determine if they refer or not to the same entity (i.e., if they *match*). Nevertheless, this would determine a huge number of comparisons, which is uselessly expensive and in many cases not even affordable due to the size of the dataset. Hence, the second issue to overcome is represented by scalability, with the need for techniques to efficiently discard as many obvious non-matches as possible while retaining useful comparisons.

The standard pipeline for entity resolution, depicted in Figure 2.2, is designed to address the two presented challenges. As previously stated in Chapter 1, it covers three main steps: *blocking* (which can be followed by an optional *block processing* step), *entity matching*, and *entity clustering*. The produced clusters of matching records are then served as input to data fusion, which produces a single representative record from each cluster through a conflict resolution function. The steps presented in Figure 2.2 are described in more detail in the following dedicated subsections.

Blocking

Blocking [Christen, 2012a; Papadakis et al., 2021b], which represents the first step in the pipeline, has the goal to make the entity resolution process scale. In fact, this phase determines the candidate pairs of records (i.e., *candidates*)

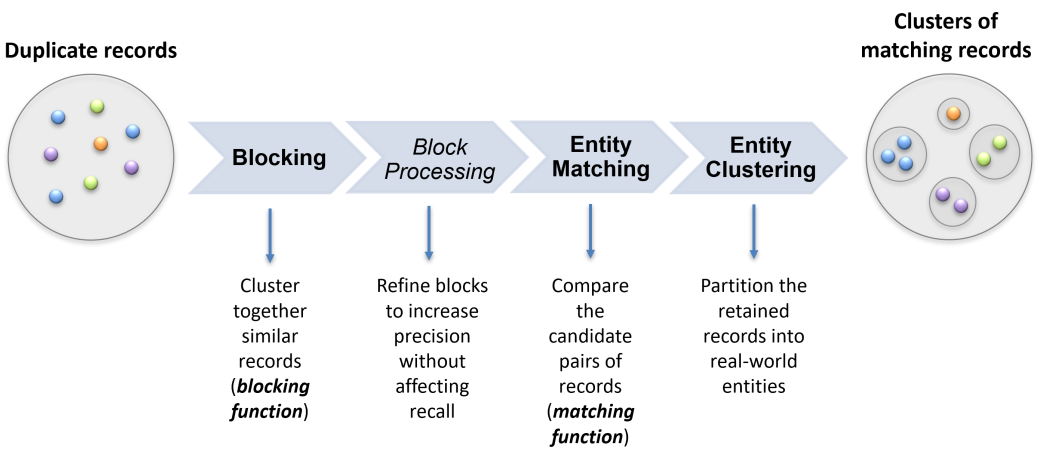

Duplicate records
Clusters of matching records
Blocking
Block Processing
Entity Matching
Entity Clustering
Cluster together similar records (blocking function)
Refine blocks to increase precision without affecting recall
Compare the candidate pairs of records (matching function)
Partition the retained records into real-world entities

Figure 2.2: The entity resolution pipeline: from a dirty dataset containing duplicates to the clusters of matching records.

that need to be checked more carefully later in the entity matching step to determine if they represent a match or not. The goal of blocking is therefore to discard from this candidate set as many obvious non-matches as possible. In particular, blocking builds clusters of similar records (i.e., *blocks*), according to the similarity criteria defined by the *blocking function*. Comparisons are then performed only among records appearing in the same block, avoiding therefore to compare pairs of records that are too dissimilar to represent a match. The blocking function is required to be computationally cheap and to produce a candidate set with a very high *recall* (i.e., the ratio of matching pairs included in the candidate set over all matching pairs present in the dataset) to retain as many matches as possible and a good *precision* (i.e., the ratio of matching pairs over all pairs included in the candidate set) to discard as many obvious non-matches as possible.

A classical example of blocking function is the widely adopted *token blocking* [Papadakis et al., 2011]. Token blocking takes into account all attributes of the considered records or a specified subset of their attributes and generates a block for each token appearing there. Tokenization is generally performed at word level, but many variants are possible, considering for instance q-grams [Augsten and Böhlen, 2013]. Each block contains therefore the identifiers of the records in which the token appears. While most generic solutions simply consider the distinct tokens (independently from the attribute in which they appear) to generate the blocks, hence adopting a *schema-agnostic* approach, some variants also differentiate among the appearances of the same token depending on the attribute in which they occur, operating therefore in a *schema-aware* or *loosely schema-aware* manner [Simonini et al., 2016].

Token blocking produces *overlapping* blocks, since a record can appear in multiple blocks (i.e., one for each of its distinct tokens). This may offer advantages in guaranteeing higher recall compared to solutions that produce *disjoint* blocks, since it enhances the probability that two matching records co-occur in at least one block. Other popular blocking techniques include for instance canopy clustering [McCallum et al., 2000], similarity joins [Barlaug, 2023b], and solutions based on TF/IDF [Paulsen et al., 2023] or even on deep learning [Thirumuruganathan et al., 2021].

Block Processing

As illustrated in Figure 2.2, blocking (especially in presence of overlapping blocks) can significantly benefit from a following *block processing* phase [Papadakis et al., 2021b], which aims to further prune useless comparisons from the produced candidate set, acting at the level of entire blocks or single pairs of records. This allows to further enhance precision while retaining recall, leading to relevant performance improvements.

In particular, *block purging* and *block filtering* operate at the block level to filter out oversized blocks, for instance the ones generated from stopwords, which determine many useless comparisons and do not add value to smaller blocks generated from meaningful tokens. While the former technique operates on all generated blocks at once, setting an upper bound to their size or cardinality (i.e., the number of produced comparisons), the latter considers for each record the blocks in which it appears and only retains the smaller ones, according to a specified percentage.

On the other hand, *meta-blocking* [Papadakis et al., 2014a; Papadakis et al., 2014b; Simonini et al., 2016; Gagliardelli et al., 2019; Gagliardelli et al., 2022a] performs pruning at comparison level, based on the construction of a *blocking graph* that represents records as nodes and inserts an edge between two nodes if the corresponding pair of records appears in the candidate set. Edges are then weighted to reflect the similarity of the two records, for instance by tracking the number of blocks in which they co-occur. After constructing the blocking graph, weights are used to prune edges and/or nodes to only retain most promising comparisons.

Entity Matching

While blocking aims to make entity resolution scale, reducing the Cartesian product of the records to a much smaller subset of promising pairs, entity matching has the goal to carefully evaluate through a *matching function* (i.e., *matcher*) these candidate pairs of records to decide if they represent a match or not. In literature, an extremely wide range of solutions for entity matching has been produced over years. For instance, the matching function can be represented by a human acting as an *oracle* [Firmani et al., 2016], or by several humans in the case of *crowdsourcing* [Gokhale et al., 2014; Das et al., 2017]. Alternatively, as often happens in industrial scenarios, it can be composed by a set of human-defined rules [Gagliardelli et al., 2020], relying for instance on some relevant patterns present in the data and on the domain knowledge to deal with homonymy and synonymy [Zecchini et al., 2020].

Solutions based on artificial intelligence proved to be very effective in entity matching, recording state-of-the-art results in many scenarios. These solutions can be based either on traditional machine learning models or on deep learning models. For instance, MAGELLAN [Konda et al., 2016; Doan et al., 2020] implements matchers based on traditional machine learning models (e.g., decision tree, random forest, SVM, etc.) and exploits as features several similarity measures computed on a set of aligned attributes. On the other hand, DEEPER [Ebraheem et al., 2018] and DEEPMATCHER [Mudgal et al., 2018] are notable examples of matchers based on neural networks [Barlaug and Gulla, 2021].

Despite registering notable results, these models need a significant amount of labeled data for their training and testing, which might require an overwhelming human effort. To mitigate this problem, different research directions have been followed, including for instance *active learning* and *transfer learning*. In particular, active learning [Settles, 2012] significantly reduces the labeling effort by carefully selecting, in an iterative process, a small amount of relevant pairs to be labeled by an oracle. This selection aims to capture the pairs of records that maximize the matcher's learning, for instance the most uncertain ones [Meduri et al., 2020]. After being labeled by the oracle, these pairs are then added to the current training set to retrain the matcher at the end of the iteration.

Outstanding results have been achieved through the application to entity matching of the *pre-training/fine-tuning* paradigm followed by BERT [Devlin et al., 2019] and its derivatives, such as DISTILBERT [Sanh et al., 2019] or ROBERTA [Liu et al., 2019]. Although initially designed in the context of natural language processing, these models were then successfully exploited in several areas of computer science. Fine-tuning these *pre-trained* deep learning models for entity matching usually requires a relatively small labeling effort and can often lead to remarkable performance, as shown by BERT itself [Paganelli et al., 2022] and by systems based on it or its derivatives, such as DITTO [Y. Li et al., 2020], which can be identified as the current state-of-the-art solution for entity matching.

More recently, the explosion of large language models led researchers to investigate their possible application as matchers. While many aspects still need to be explored, for instance the impact of prompting [Sisaengsuwanchai et al., 2023], preliminary results presented in literature show enormous potential in entity matching [Narayan et al., 2022; Peeters and Bizer, 2023], often achieving notable results even in the zero-shot learning case.

Finally, together with the wide adoption of complex artificial intelligence models for entity matching, the latest years saw the first research results about the *explainability* of such matchers. Explainable entity matching aims to understand the reasons behind the classifications performed by a matcher, allowing to understand if it is considering relevant features for its decisions or if it is driven by misleading aspects. At the same time, this allows to *debug* the training data, in case of artificial intelligence models, spotting for instance the presence of wrongly labeled pairs (or underrepresented cases) that influence negatively the model decisions. Most of the proposed techniques [Di Cicco et al., 2019; Baraldi et al., 2021; Barlaug, 2023a] are inspired by LIME [Ribeiro et al., 2016], a solution for *local* explainability [Guidotti et al., 2019] that relies on surrogate interpretable models (e.g., linear classifiers) to understand the weight of different features behind a single decision taken by a black box model. This research direction evolved by considering *counterfactual explanations* [Teofili et al., 2022; Wang and Y. Li, 2022], i.e., the minimum change to a pair of records that would modify its classification. Nevertheless, all these approaches present the problem of carefully selecting representative pairs for a *global* overview of the matcher behavior [Laskowski and Sold, 2023]. More recently, intrinsically interpretable entity matching systems such as WYM [Baraldi et al., 2023] have been explored too.

Entity Clustering

Whatever the choice of the matching function, it presents a binary nature, taking decisions on a single pair of records at a time. This approach can lead to the insurgence of inconsistencies: for instance, considering the records A, B, and C, it might be possible that the matcher classifies the pairs A-B and A-C as matches, while B-C is denoted as a non-match. The goal of entity clustering is exactly to solve these inconsistencies, producing from single matches a set of consistent clusters of records that refer to the same entity, which are then served to data fusion for generating the cleaned version of the dataset. A typical solution is for instance *transitive closure* (i.e., considering the connected components in a graph where nodes represent records and edges denote pairs classified as matches), that would mark also the pair B-C as a match. Nevertheless, transitive closure is a very simplistic solution that might lead to the generation of heterogeneous clusters. Hence, several more sophisticated algorithms have been designed and evaluated over years to produce clusters of matching records with a better accuracy [Hassanzadeh et al., 2009].

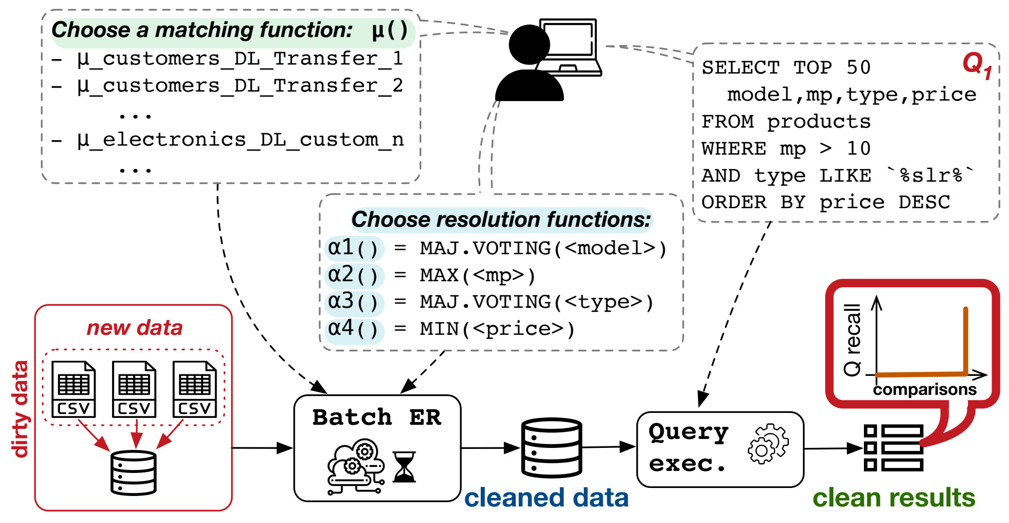
End-to-End Entity Resolution Ecosystems

As illustrated above, entity resolution is composed of multiple subtasks with dedicated roles: blocking, to make the problem scale; entity matching, to determine if a pair of records represents a match or not; entity clustering, to ensure the consistency of the decisions taken by the matcher. Since managing each step independently using dedicated libraries may be tricky, some *end-to-end* entity resolution ecosystems have been conceived to support the entire entity resolution pipeline. For instance, MAGELLAN [Konda et al., 2016; Doan et al., 2020] not only supports machine learning models for entity matching, but also simple techniques to perform blocking in advance, such as *overlap blocking*. A prominent attention to the blocking phase is guaranteed instead by JEDAI [Papadakis et al., 2019; Papadakis et al., 2020], which implements most of the state-of-the-art algorithms to perform blocking and block processing, in particular meta-blocking. JEDAI also supports a wide range of entity clustering techniques and, especially in its recent Python adaptation named PYJEDAI [Nikoletos et al., 2022], several matchers of different types.

2.2 Beyond Batch Entity Resolution

Typically, entity resolution is employed as an expensive *offline* cleaning step *before* using the data. Hence, the entire entity resolution pipeline has to be applied on the entire data at hand to obtain the cleaned version of the data, which can be later used in downstream tasks. We refer to this approach as *batch* entity resolution, since the clean entities are made available to the practitioner all at once after the entire dataset has been cleaned. Yet, in many practical scenarios this might not be convenient. Let us consider for instance the following real-world example.

Ellen is a data scientist building a machine learning model to predict the price of SLR (Single-Lens Reflex) cameras. The scenario presents three relevant features. (i) *She has limited time to add more data to her dataset and clean it; also, new data might be arriving periodically.* (ii) *The data might contain duplicates. For performing entity resolution, she already has a matching function to choose (or more than one to try) for the data at hand, for instance a machine learning model trained on the data she already has and/or*


Choose a matching function: µ()
- µ_customers_DL_Transfer_1
- µ_customers_DL_Transfer_2
...
- µ_electronics_DL_custom_n
...
Choose resolution functions:
new data
dirty data
CSV
CSV
CSV
Batch ER
cleaned data
Query exec.
Q recall
comparisons
clean results
Q1

```
α1() = MAJ.VOTING(<model>)
α2() = MAX(<mp>)
α3() = MAJ.VOTING(<type>)
α4() = MIN(<price>)
```

```
SELECT TOP 50
  model,mp,type,price
FROM products
WHERE mp > 10
AND type LIKE `%slr%`
ORDER BY price DESC
```

Figure 2.3: The traditional *batch* entity resolution pipeline: the data scientist specifies how to clean the data with entity resolution; once cleaned, she issues the query.

exploiting transfer learning and/or ad-hoc rules. Further, she expresses rules for resolving the conflicts in the attribute values of the clusters of matching records, e.g., AVG(`price`*), MAX(*`resolution`*), etc. These rules allow to perform data fusion to obtain the final clean entities.* (iii) *She has business priorities: it is better to have clean data for expensive cameras than for inexpensive ones, and only modern cameras with a minimum resolution of 10 MP have to be considered. She can express this with a query.*

Figure 2.3 shows how Ellen specifies data extraction and cleaning, i.e., the SQL query Q_1 that selects the entities she is interested in (priority on expensive cameras given by the ordering predicate), the matching function, and the resolution functions to be used in data fusion.

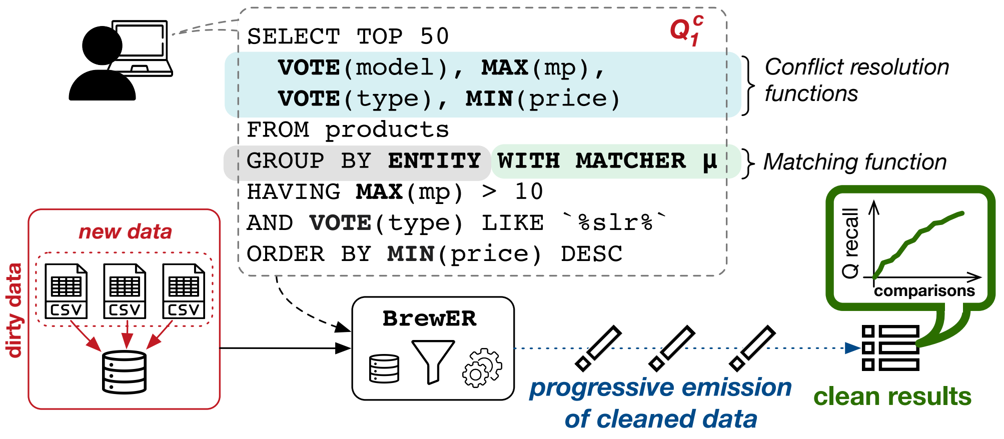
As depicted in this example, oftentimes practitioners (e.g., data scientists) have a specific task at hand characterized by: *(i)* an *information need*: only a portion of the entities is relevant to their task, hence to clean the entire data just to run a selective query on it is a waste of resources; *(ii) time constraints*: time can be limited and decisions based on the data have to be made quickly; time can be limiting when new data arrives or changes with high frequency and users want to quickly explore it with queries (e.g., top-k queries). Let us therefore consider how the batch approach would perform in such scenarios.

With a traditional framework, Ellen performs entity resolution on the *entire data at her disposal by applying matching and resolution functions. After the entire data is cleaned, she can issue the query and explore or query the result (Figure 2.3). In such a scenario, entity resolution is the bottleneck due to:* (i) *its inherently quadratic complexity, which blocking and filtering techniques can only alleviate;* (ii) *the cost of matching functions, as state-of-the-art matchers involve expensive operations based on string-similarity measures computation or deep neural network models.*

This approach is time-consuming. In fact, to check whether a new source contains useful data for her analysis with a query, she has to first clean it completely: all entities are resolved and then filtered to produce results emitted in batch. Further, for debugging the entity resolution pipeline with the data at hand (e.g., to check if the matcher she is employing is performing well for expensive SLR cameras), she cannot stop the entity resolution process after receiving a handful of the entities to inspect and then resume the processing: those entities might not be relevant for the query or might be partially resolved, hence yielding incorrect results. Alternatively, she would have to manually select records from the dataset to test the entity resolution pipeline, which is time-consuming as well.

Thus, it would be beneficial to *prioritize* the cleaning efforts on the entities according to their relevance for the practitioner's task. In the existing literature, two main research directions have been explored to go beyond the batch entity resolution pipeline: *(i) progressive* approaches, which aim to perform entity resolution progressively by prioritizing candidates according to their matching probability, but can guarantee neither the correctness of the results in case of early termination nor support user-defined priorities; *(ii) query-driven* approaches, which aim to perform entity resolution only on the portion of the dirty data that might be useful to answer the issued query, but are still designed to operate in a batch manner, hence do not support progressiveness. The comparison with these approaches is explored in detail in Section 2.3.

We propose therefore the BrewER framework, which evaluates SQL SP (*Selection* and *Projection*) queries on dirty data, and returns results as if they were issued on clean data (as shown in Figure 2.4). The main feature of BrewER is to perform entity resolution *progressively*, guided by an ORDER BY clause, to incrementally return the most relevant results to the data scientist. BrewER avoids as much as possible matching and resolving entities that are not part of the final result, and it inherently supports top-k queries, as well as *stop-and-resume* execution. BrewER introduces a special "GROUP BY ENTITY WITH MATCHER [matcher of choice]" operator, which is interpreted as a "group by entity" statement, i.e., knowing that matching records should be grouped according to the selected matcher.



```
SELECT TOP 50
   VOTE(model), MAX(mp),
   VOTE(type), MIN(price)
FROM products
GROUP BY ENTITY WITH MATCHER μ
HAVING MAX(mp) > 10
AND VOTE(type) LIKE `%slr%`
ORDER BY MIN(price) DESC
```

Figure 2.4: The entity resolution on-demand pipeline, implemented using BrewER: the data scientist specifies how to clean the data within the query.

As shown in Figure 2.4, with BrewER *Ellen just needs to rewrite her query (Q_1 in Figure 2.3) by employing a special GROUP BY statement and moving the selection statements into the HAVING clause (predicating on each group, i.e., each entity). She also specifies the conflict resolution functions for data fusion within the SQL query, as aggregate functions. Note that Q_1^c in Figure 2.4 and Q_1 in Figure 2.3 are equivalent if issued on clean data. Then,* BrewER *executes the query directly on the dirty data, applying entity resolution progressively on the right portion of the data to yield correct results incrementally.*

Ellen receives the first entities in a fraction of the time required by existing entity resolution frameworks. This allows her to explore new data without completely cleaning it, and to maximize the entity resolution efforts on the entities she actually needs for her task. Furthermore, she can stop the execution at any time with the guarantee that the results produced so far are correct; then, she can resume the query evaluation at her need. Thus, she can inspect the result of the entity resolution process for entities of interest and debug it, if needed. Finally, BrewER *keeps track of both executed comparisons and resolved entities, to avoid recomputing the same operations when multiple queries are issued on the same data.*

Another example is the stock market trading scenario, where an entity matching algorithm on a high frequency financial news feed may have very limited time to match companies' records and yield useful information [Whang et al., 2013]. Further, typically only a subset of the entities is relevant for each operation and a priority may be defined, such as the trading volume or other financial metrics. Moreover, our proposed approach is suitable for tackling a major challenge for data lake management systems [Nargesian et al., 2019]: to support on-demand extraction and cleaning as part of the data integration pipeline and on-demand query answering. Similarly, on-demand data transformation that returns results in a timely manner is a fundamental requirement of ELT pipelines, especially when combined with top-k queries to debug transformations [Cohen et al., 2009].

2.3 Comparison with Related Work

This section presents an extensive discussion about the major differences between entity resolution on-demand and the two main approaches previously designed to overcome the limitations of the traditional batch approach, i.e., *progressive* entity resolution and batch *query-driven* entity resolution.

Progressive Entity Resolution

Madhavan et al. firstly proposed a *progressive* (a.k.a. *pay-as-you-go*) data integration system in 2007, implemented for Google Base, to progressively integrate as much Web data as possible as it runs, given a limited amount of time and resources [Madhavan et al., 2007]. The progressive approach has been then employed for schema alignment [Das Sarma et al., 2008] and entity resolution [Whang et al., 2013]. In particular, to perform entity resolution, oftentimes the resources are limited (e.g., computational power or human time to debug the entity resolution pipeline), or the data has to be elaborated within a certain time to be valuable for the downstream application consuming it. To address this challenge, existing progressive entity resolution methods [Whang et al., 2013; Papenbrock et al., 2015a; Firmani et al., 2016; Simonini et al., 2018; Galhotra et al., 2021] try to evaluate candidate matches by their likelihood of being actual matches (typically estimated through a proxy measure derived from a blocking strategy), so to discover as many matches as possible, as quickly as possible. Thus, a progressive entity resolution method incrementally adds records to an entity cluster, which might remain incomplete until the entire data is processed. Hence, a SQL SP query cannot be simply issued at any time on the output of a progressive method. In fact, its result might be incorrect: the representative records may have values derived applying resolution functions on a partially identified entity cluster, thus yielding possibly incorrect values. BrewER aims at finding and resolving complete entity clusters, whose representative records satisfy a SQL SP query; moreover, it does that progressively, while following an ORDER BY predicate. Both BrewER and existing progressive methods deal with a *single-type entity* dataset at a time, i.e., a dataset with a unique schema. In Section 2.7.2, we further discuss and experimentally compare the progressive entity resolution methods and BrewER.

Batch Query-Driven Entity Resolution

To the best of our knowledge, the closest works to our own are QDA [Altwaijry et al., 2013] and QuERy [Altwaijry et al., 2015]. QDA takes as input a single block B and a selection predicate P, then analyzes which record pairs do not need to be compared to identify all entities in B that satisfy P. QDA is not designed for a progressive execution and to support ORDER BY clauses. Moreover, QDA requires to apply the resolution functions to the output of each match, hence it cannot support some very common functions that consider more than two values, such as AVG and VOTE (i.e., *majority voting*). BrewER is blocking-agnostic (i.e., it is not limited to one block) and supports a wider class of resolution functions, including AVG and VOTE. QuERy supports SQL SPJ queries by introducing two special selection and join operators, which are called *polymorphic* as they accept as input not only records (as regular operators), but also objects representing blocks (called *sketches*). A sketch is a concise representation of all the potential representative records that a block may yield (i.e., all possible outcomes of the cleaning of a block). For instance, a sketch employs a *range* data type to represent numerical attributes and a *set of hashed values* to represent categorical attributes. To evaluate a query, QuERy builds a query tree (i.e., an execution plan) with the *polymorphic operators* for a dataset composed of clean records and sketches. When the query tree is executed, if a sketch reaches the topmost operator (i.e., passes all predicates), the corresponding block has to be cleaned since it can contain useful entities, otherwise it is discarded. The representative records (i.e., the cleaned entities) yielded from the cleaned block are then pushed back into the query tree to be re-evaluated, to check if they actually pass the predicates. QuERy's main limitation is that its polymorphic operators work at the block level and cannot define the progressiveness of the entity resolution execution within each block. BrewER could be employed within each block to reduce the number of comparisons that are evaluated and progressively pass resolved entities up to the query tree. Then, for a complete integration of BrewER and ORDER BY clauses, a *polymorphic sort* operator should be designed to compare and sort records and sketches. We do not investigate further the integration of QuERy and BrewER here. Finally, the idea of *query-driven* entity resolution has been explored also for answering *keyword queries* [Sartori et al., 2016; L. Zhu et al., 2018]. A previous preliminary work [Pietrangelo et al., 2018] explored how to yield an approximate progressive result to a *keyword query* over a dirty dataset. BrewER guarantees an exact (i.e., not approximate) result instead and supports SQL SP queries.

2.4 Preliminaries

This section introduces the formalism used in the detailed description of our algorithm. In particular, Section 2.4.1 formalizes the general entity resolution model, recalling many concepts introduced in Section 2.1, while Section 2.4.2 provides a formal definition to the entity resolution on-demand model implemented by BrewER.

2.4.1 Entity Resolution Model

We consider a dirty dataset $\mathcal{D}$ with schema $\mathcal{D}[A_1, ..., A_m]$. Each attribute A_j has a domain (or *type*) T_{A_j} of values that the records can assume. A record $r \in \mathcal{D}$ is represented as a tuple $r = (id, r[A_1], ..., r[A_m])$, where id is a unique identifier for each r, and $r[A_j] \in \{T_{A_j} \cup \emptyset\}$ is a projection to the value that the record assumes for the j-th attribute (note that null values are admitted). Different records in a dataset that *belong* (i.e., refer) to the same real-world *entity* are called *matching records* (or simply *matches*). Entity Resolution (ER) aims to identify the disjoint clusters of records representing the entities of $\mathcal{D}$ and to synthesize a single *representative record* $\varepsilon = (id, \varepsilon[A_1], ..., \varepsilon[A_m])$ for each cluster, so to produce $\mathcal{D}^c$, the *cleaned* version of $\mathcal{D}$.

We adopt a standard entity resolution framework [Benjelloun et al., 2009; D. Deng et al., 2019; Doan et al., 2020] employing a *matching function* for determining the matching records that form entity clusters and *conflict resolution functions* for consolidating ambiguous attribute values within each cluster (i.e., performing *data fusion*). We also support optional *blocking*, which is often employed to scale entity resolution by avoiding comparing obvious non-matching pairs of records [Papadakis et al., 2021b].

Matching Function

A *matching function* (a.k.a. *matcher*) is a binary function $\mu : \mathcal{D} \times \mathcal{D} \to \{True, False\}$ that takes as input two records, compares them, and decides whether they are matches or not. We do not assume the matching function to be transitive, i.e., if $\mu(r_x, r_y) = True$ and $\mu(r_y, r_z) = True$, it might be that $\mu(r_x, r_z) = False$. To design a transitive matching function is difficult in practice [Benjelloun et al., 2009]. Yet, we consider the matches to be transitive, otherwise results would be inconsistent, e.g., declaring $\langle r_1, r_2 \rangle$ and $\langle r_2, r_3 \rangle$ as matching pairs while declaring $\langle r_1, r_3 \rangle$ as non-matching.

Our framework is matcher-agnostic: it can support any kind of matching function, such as a DNF of similarity join predicates on multiple attributes [Gagliardelli et al., 2020], a human judging the pairs of records on a crowdsourcing platform [Firmani et al., 2016; G. Li, 2017], an unsupervised matcher based on generative models [Wu et al., 2020], or a complex deep learning model exploiting pre-trained language models [Y. Li et al., 2020] or transfer learning [Loster et al., 2021]. Our framework allows indicating the matching function for a particular entity resolution task within a SQL query denoted by μ^Q.

Conflict Resolution Functions

The *conflict resolution functions* (or simply *resolution functions*) transform a cluster of records into a single record. Records belonging to the same entity cluster often have inconsistent values for their attributes (e.g., camera records referring to the same entity may have different prices, names, etc.). Thus, for each attribute, a resolution function is applied to remove conflicts: it takes as input a multiset of attribute values and returns a single value.

We declare a resolution function for an attribute A_j through a SQL *aggregate function* α_j. Given a cluster of records $\mathcal{E} = \{r_1, ..., r_k\}$ representing a single entity, each α_j takes as input the list of values that A_j assumes in those records and returns a single value $\varepsilon[A_j] \in T_{A_j}$. The aggregate functions supported in our framework are: MIN (minimum), MAX (maximum), AVG (average), and user-defined *bounded aggregations* (defined later in the current section), such as MEDIAN and VOTE (a.k.a. *majority voting*). The choice of this set of functions was driven by two considerations: *(i)* they cover most real-world use cases; *(ii)* they can be naturally declared as part of SQL queries.

Blocking

Comparing all pairs of records in $\mathcal{D}$ has a quadratic complexity and, typically, the matching function is expensive to compute: state-of-the-art functions involve either string-similarity computation [Doan et al., 2020] or deep neural networks [Y. Li et al., 2020]. *Blocking* is employed to partition $\mathcal{D}$ in *blocks*, i.e., partitions of records, and limits the all-pairs comparison to records within each block [Christen, 2012a; Papadakis et al., 2021b]. Given a record r, we call *candidate set* the set of records that appear together with it in blocks.

Each record in it is called *candidate match* or simply *candidate*.

Our framework is blocking-agnostic, meaning that any blocking strategy (or even no blocking strategy) can be employed, including unsupervised techniques [O'Hare et al., 2019; Papadakis et al., 2020; Thirumuruganathan et al., 2021], as experimentally shown in Section 2.7.5.

Query-Agnostic (Traditional) Entity Resolution Algorithms

We call *traditional* entity resolution algorithms those algorithms that employ matching and conflict resolution functions in a query-agnostic way, i.e., without filtering for a specific part of the data and without order preferences. According to this definition, traditional algorithms may follow any match-resolve strategy, such as *batch* entity resolution [Christen, 2012b], which performs blocking, applies the matching function in random order, and finally performs the conflict resolution, or *progressive* entity resolution [Whang et al., 2013; Papenbrock et al., 2015a; Firmani et al., 2016; Simonini et al., 2018; Galhotra et al., 2021], presented in Section 2.3.

Record Bounds and Bounded Aggregation

Given a record r, its candidate set, and an aggregate function α_j for a numeric attribute A_j, we call *lower bound* and *upper bound* of r the minimum and maximum value that the entity to which r belongs can assume for the attribute A_j, respectively.

When an aggregate function α_j yields a consolidated value $\varepsilon[A_j]$ for a set of matches $\mathcal{E}$ that is always $\varepsilon[A_j] \in [min(V^{\mathcal{E}}_{A_j}), max(V^{\mathcal{E}}_{A_j})]$, we call it a *bounded aggregation*; otherwise, we call it an *unbounded aggregation* (e.g., SUM, which can produce a final value $\varepsilon[A_j]$ that is greater than the maximum value of $V^{\mathcal{E}}_{A_j}$). We consider only MIN, MAX, AVG, MEDIAN, and VOTE for our examples or experiments, but BrewER inherently supports any user-defined bounded aggregation (UDF). We do not study unbounded aggregation instead.

Further, we distinguish between *fixed* and *free* bounded aggregate functions for numeric attributes. Given a numeric attribute A_j, a *fixed* aggregate function can yield only values $\varepsilon[A_j] \in V^{\mathcal{E}}_{A_j}$, i.e., the value that the resolved entity ε assumes in A_j is among the values that its records assume for A_j. Examples of fixed aggregate functions are MIN, MAX, and VOTE. A *free* aggregate function, on the other hand, can generate $\varepsilon[A_j] \in [min(V^{\mathcal{E}}_{A_j}), max(V^{\mathcal{E}}_{A_j})]$, i.e., the value assumed by A_j in the resolved entity ε can be a *new* value not among the values that its records assume in A_j, yet bounded from them. An example of free aggregate function is AVG.

```
    SELECT [TOP $k$] $\langle \alpha_j(A_j) \rangle$
      FROM $\mathcal{D}$
    [WHERE $\varphi$]
  GROUP BY $ENTITY$ WITH MATCHER $\mu$
   [HAVING $\langle \alpha_j(A_j)$ {LIKE|IN|<|≤|>|≥|=} $const \rangle$]
[ORDER BY $\alpha_j(A_j)$ [ASC|DESC]]
```

Figure 2.5: Query syntax in BrewER.

2.4.2 Entity Resolution On-Demand Model

This section first introduces the type of queries supported by our framework, then describes the features of an entity resolution on-demand algorithm.

Supported Queries

BrewER supports SQL SP queries with an ordering predicate. Like existing progressive methods (Section 2.3) and most state-of-the-art entity resolution frameworks, such as Magellan [Konda et al., 2016; Doan et al., 2020], JedAI [Papadakis et al., 2019; Papadakis et al., 2020], DeepMatcher [Mudgal et al., 2018], Ditto [Y. Li et al., 2020], or Tamr [Stonebraker et al., 2013], we focus only on *single-type entity* datasets (e.g., electronic goods), i.e., BrewER applies entity resolution on dirty datasets that can be represented with a unique schema. In such a scenario, SP predicates and ordering predicates are sufficient for users to express their information needs and priorities, respectively. The JOIN operator would be useful when dealing with *multi-type entity* datasets, i.e., when entity resolution is applied jointly on multiple dirty datasets with different schemata [Whang and Garcia-Molina, 2012]. We aim at investigating this dimension in future work.

Figure 2.5 presents the syntax supported in BrewER. The GROUP BY ENTITY clause declares that the query should return results aggregated by entities. In BrewER, "ENTITY" is a reserved word and must be combined with a matching function μ. The resolution functions are specified in the SELECT clause as a list $\langle \alpha_j(A_j) \rangle$, where each element is an aggregate function α_j combined to an attribute A_j. The WHERE clause serves just as a filter applied directly to the initial dirty data $\sigma_\varphi(\mathcal{D})$, i.e., it filters records *before* the cleaning. From now on, for simplicity, φ is omitted, being a filter applied to the dirty data independently of the entity resolution process. The filtering on the entities *after* the entity resolution process is defined in the HAVING clause, which predicates over aggregate values of the groups (i.e., values of the entities). In the HAVING clause, we currently support numeric comparisons

($<$, $\leq$, $>$, $\geq$, $=$) for numeric attributes and dates, and string comparisons ($=$, LIKE, IN) for textual attributes. Finally, we currently support ordering for a single attribute.

In BrewER, a valid query Q^c has the structure presented above in Figure 2.5. With Q we denote the corresponding query for cleaned data, where: *(i)* the GROUP BY statement is removed; *(ii)* the HAVING predicates are expressed as WHERE conditions; *(iii)* no aggregation is specified in the selection statement; *(iv)* there is an ORDER BY condition on the same attribute of Q^c. In practice, Q^c issued on a dirty dataset $\mathcal{D}$ yields the same results of Q issued on the cleaned dataset $\mathcal{D}^c$ (cleaned with the same matcher and resolution functions of Q^c). Examples of Q and Q^c are shown in Figures 2.3 and 2.4, respectively.

The ORDER BY clause allows to benefit from the progressive emission of the entities. Yet, a user can define a query without using it. In such a case, BrewER chooses a random (even textual) attribute for the ordering. Similarly, a user may express a query without a selection predicate. In this case, all entities are emitted progressively, following the ORDER BY clause, in a *pay-as-you-go* fashion.

Entity Resolution On-Demand Algorithm

Given a (dirty) dataset $\mathcal{D}$ and a BrewER SQL query Q^c, we want an algorithm to guarantee a correct partial result when the execution is terminated early. That algorithm should perform entity resolution progressively, following the query Q^c and its ORDER BY clause, and not up-front on the entire data (i.e., *on-demand*). In fact, traditional entity resolution algorithms do not guarantee the correct result in case of early termination: some of the emitted entities may not be completely resolved, thus possibly sorted in the wrong order and/or erroneously retained/discarded due to unresolved inconsistencies of their values. Thus, an *entity resolution on-demand algorithm* allows to significantly save computational resources and time if the user wants to stop the execution after inspecting the first k emitted records. Hence, top-k queries and *stop-and-resume* execution are inherently supported.

We now formally define an entity resolution on-demand algorithm. We denote with $Q(\mathcal{D}^c)$ the results of a SQL SP query Q with an ORDER BY clause issued on $\mathcal{D}^c$, which is the cleaned version of the dirty dataset $\mathcal{D}$ obtained by using a traditional entity resolution algorithm. The corresponding valid version of the query Q in BrewER is Q^c, i.e., a query written according to the syntax shown in Figure 2.5.

Definition 2.4.1 (Entity Resolution On-Demand Algorithm). Given a dirty dataset $\mathcal{D}$ and a SQL SP query Q with an ORDER BY clause, an *entity resolution on-demand algorithm* to evaluate Q^c (i.e., the BrewER version of Q) on $\mathcal{D}$ satisfies the following conditions:

1. *Correctness*: Let t be an arbitrary target time at which the results are needed: $Q^c_t(\mathcal{D}) \subseteq Q(\mathcal{D}^c)$ and $Q^c_t(\mathcal{D})$ is correctly sorted according to the ORDER BY condition. Typically, t is significantly smaller than the time needed to produce $\mathcal{D}^c$ in its entirety.

2. *Monotonicity*: $Q^c_{t_1}(\mathcal{D}) \subseteq Q^c_{t_2}(\mathcal{D})$ for any $t_1 < t_2$.

3. *Equivalence*: If the traditional entity resolution algorithm and the entity resolution on-demand algorithm both have enough time to terminate, they produce the same results for the query, i.e., $Q^c_{t_\infty}(\mathcal{D}) \equiv Q(\mathcal{D}^c)$.

Notice that a traditional entity resolution algorithm does not satisfy Conditions 1 and 2. Let us consider a progressive entity resolution algorithm, a query Q^c, and a dirty dataset $\mathcal{D}$, and assume that the representative record ε of a cluster of matching records $\{r_1, r_2, r_3\}$ does not satisfy Q^c. Let us further assume that if we resolve only $\{r_1, r_2\}$ the resulting (incomplete) representative record ε' satisfies Q^c (a common scenario with real-world data). Now, say that after running for a time t_1, the progressive algorithm identifies only two matches: $\{r_1, r_2\}$. So, if we interrupt the execution at t_1 and issue Q^c, $\varepsilon_{t_1} \equiv \varepsilon'$ satisfies Q^c and is erroneously emitted as a result. Hence, $Q^c_{t_1}(\mathcal{D}) \nsubseteq Q(\mathcal{D}^c)$ and thus *correctness* is not satisfied. Then, if we run the progressive algorithm until the discovery of the remaining match r_3 (i.e., at time $t_2 > t_1$) and issue again Q^c, $\varepsilon_{t_2} \equiv \varepsilon$ does not satisfy Q^c. Hence, $Q^c_{t_1}(\mathcal{D}) \nsubseteq Q^c_{t_2}(\mathcal{D})$ and thus *monotonicity* is not satisfied.

2.5 Entity Resolution On-Demand

The high-level design of BrewER is depicted in Figure 2.6. BrewER is an extensible framework, enabling users to plug-in their favorite libraries for binary matching functions, e.g., DeepMatcher [Mudgal et al., 2018] or Ditto [Y. Li et al., 2020], and blocking, e.g., Magellan [Doan et al., 2020]. Then, BrewER takes care of the entity resolution on-demand execution of the user's query, as explained in the remainder of this section. To avoid re-comparing candidate pairs with expensive matching functions, the lists of matching and non-matching records are maintained in a database, for each matching function employed by the users (if the matching function changes, the matches change as well). Thus, BrewER can retrieve, exploit, and update those lists when executing a new query. Users can also choose to

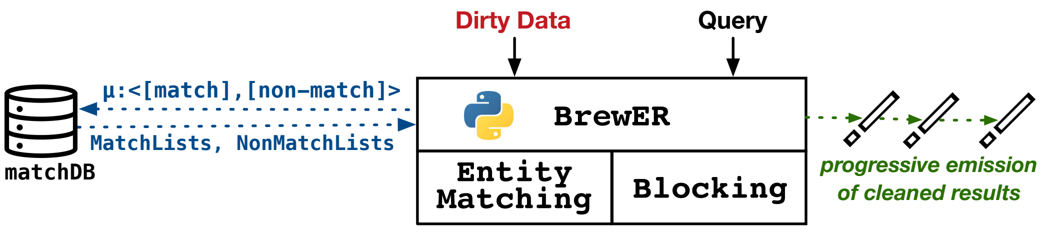


Figure 2.6: An overview of BrewER.

store only final resolved entities to save space. In this case, the resolution functions cannot change across queries.

At the core of BrewER lies our entity resolution on-demand algorithm, introduced in Section 2.5.1, then formally presented in Section 2.5.2.

2.5.1 Algorithm Overview

The ideal entity resolution on-demand algorithm would start by cleaning the first entity in $\mathcal{D}$ that should be emitted for the query Q issued on cleaned data $\mathcal{D}^c$, and emit that entity as its first result. Then, it would start over with the *next entity that should be cleaned and emitted* for Q, and so forth.

To design a practical entity resolution on-demand algorithm, we first define *seed records*, which are the records that guide our algorithm in seeking the next entity that should be cleaned and emitted. We also define a *seed query*, i.e., the query to extract the seed records. The general idea is to insert the seed records and their candidate matches in a priority queue, which is exploited for ordering the entities (to which the seed records belong) that satisfy Q^c, while cleaning them.

Seed Query

We first consider the case where fixed aggregate functions are employed, to complete then the discussion with free aggregate functions.

Considering a valid BrewER query Q^c employing only fixed aggregate functions and its corresponding query Q for cleaned data, we observe that if all records in a cluster of matches do not satisfy any of the selection conditions of Q, then that cluster cannot yield an entity that is part of the result of Q^c. Thus, given a set of candidate matches, if none of the involved records satisfies at least one of the selection conditions of Q, those comparisons can be avoided. On the other hand, if there exists at least one record r_s satisfying one of the selection conditions of Q, those comparisons should be considered. We call r_s a *seed record* (or simply *seed*).

Q_1

```
SELECT TOP 50
  model,mp,type,price
FROM products
WHERE mp > 10
AND type LIKE `%slr%`
ORDER BY price DESC
```

(a) Q_1

Q_1^c

```
SELECT TOP 50
  VOTE(model), MAX(mp),
  VOTE(type), MIN(price)
FROM products
GROUP BY ENTITY WITH MATCHER μ
HAVING MAX(mp) > 10
AND VOTE(type) LIKE `%slr%`
ORDER BY MIN(price) DESC
```

(b) Q_1^c

Q_1^{seed}

```
SELECT *
FROM products
WHERE mp > 10
OR type LIKE `%slr%`
```

(c) Q_1^{seed}

Figure 2.7: The query to be issued on clean data (a), a valid BrewER query to be issued on dirty data (b) and its seed query (c).

Consider now a valid BrewER query Q^c with only a free aggregate function, e.g., a query with the condition "HAVING AVG(mp) = 10" issued on the dataset of Figure 2.8a. It may occur that no record satisfies the corresponding selection condition in Q, i.e., "WHERE mp = 10", but a cluster of matching records actually has an average of values for the attribute mp equal to 10. The process for fixed aggregate functions would not identify a seed and we would miss a correct result. Yet, a free aggregation is also a bounded aggregation, so we can discard any record r_i if $\theta_{\texttt{mp}} \notin [min(V_{\texttt{mp}}^{C_i}), max(V_{\texttt{mp}}^{C_i})]$, where $V_{\texttt{mp}}^{C_i}$ is the set of values assumed by the candidates C_i of r_i (with $r_i \in C_i$) for the attribute mp, and $\theta_{\texttt{mp}}$ is the parameter of the selection clause (i.e., $\theta_{\texttt{mp}} = 10$). This is possible because the entity to which r_i belongs cannot assume a value for mp outside the range $[min(V_{\texttt{mp}}^{C_i}), max(V_{\texttt{mp}}^{C_i})]$. Thus, for a free aggregate function on an attribute A_j, r_i is a seed record if: *(i)* for the equality operator, $\theta_{A_j} \in [min(V_{A_j}^{C_i}), max(V_{A_j}^{C_i})]$; *(ii)* for $>$ (or $<$) inequality operator, $r_i[A_j] > \theta_{A_j}$ (or $r_i[A_j] < \theta_{A_j}$).

The *seed query* Q^{seed} from a BrewER query Q^c yields the *seed set*, i.e., the set of all seed records. It is obtained with a projection of all attributes of $\mathcal{D}$ and a selection composed of the logical disjunction of the *set of basic predicates* $\mathcal{P}$ derived from the HAVING clause of Q^c as follows. If the HAVING clause of Q^c involves a fixed aggregate function, its corresponding selection predicate ϕ of Q is added to $\mathcal{P}$. If the HAVING clause of Q^c involves a free aggregate function for the attribute A_j: *(i)* for the equality operator, we add to $\mathcal{P}$ a predicate ϕ of the form θ_{A_j} BETWEEN$(min(V_{A_j}^{C_i}), max(V_{A_j}^{C_i}))$; *(ii)* for $>$ (or $<$) inequality operator, we add to $\mathcal{P}$ a predicate ϕ of the form $A_j > \theta_{A_j}$ (or $A_j < \theta_{A_j}$). No ordering is needed for the seed query. We use the logical disjunction, even for conjunctive queries (e.g., Figure 2.7), since the seed records can match and yield any entity that satisfies Q^c, although each of them individually may not satisfy all predicates of Q. Hence, the seed query is defined as $Q^{seed} = \sigma_{\bigvee \phi \in \mathcal{P}}(\mathcal{D})$.

	id	brand	model	type	mp	price
ε_1	r_1	canon	eos 400d	dslr	10.1	185.00
	r_2	eos canon	rebel xti	reflex	1.01	115.00
	r_3	canon	eos 400d	dslr	10.1	165.00
ε_2	r_4	nikon	d-200	-	-	150.00
	r_5	nikon	d200	dslr	10.2	130.00
ε_3	r_6	nikon	coolpix	compct	8.0	90.00
ε_4	r_7	canon nikon olympus	olypus-1	dslr	-	90.00

(a) A dirty dataset.

ORDER BY **AVG**(PRICE) DESC

	(VOTE)	(VOTE)	(MAX)	**(AVG)**
id	model	type	mp	price
ε_1	eos 400d	dslr	10.1	155.00
ε_2	d-200	dslr	10.2	140.00

(b) The result for Q_1^c of Figure 2.7b, with $\alpha($`price`$) \equiv$ AVG(`price`) instead of MIN(`price`), issued on the dirty dataset.

ORDER BY **MIN**(PRICE) DESC

	(VOTE)	(VOTE)	(MAX)	**(MIN)**
id	model	type	mp	price
ε_2	d-200	dslr	10.2	130.00
ε_1	eos 400d	dslr	10.1	115.00

(c) The result for Q_1^c of Figure 2.7b issued on the dirty dataset.

Figure 2.8: A dirty dataset (a) and clean query results (b and c) for different aggregate functions on the attribute `price`.

Seeds and Blocking

When blocking is employed[2], BREWER computes the transitive closure of all candidate pairs by merging blocks that overlap and stores the resulting connected components of records in an auxiliary data structure, called *component list*. A *block index* is maintained as well: an index where each connected component is a key that points to the lists of blocks that have been merged to yield that component. Then, BREWER removes from the *component list* all the components that do not contain any seed record, since they cannot yield any result for Q^c. Moreover, the set of basic predicates $\mathcal{P}$ (defined above) can be exploited to filter out further components that do not lead to any useful entity for answering Q^c.

Consider for instance Q_1^c of Figure 2.7b: if a component does not contain any record that has *slr* in its `type` attribute, then it cannot yield any entity

[2]If blocking is not employed, the entire dataset is still considered as a single block.

satisfying Q_1^c, even if the predicate on the `mp` attribute is satisfied. Hence, for conjunctive queries, BrewER builds a query Q_i^b for each i-th predicate in $\mathcal{P}$. So, if a query Q_i^b applied to a component returns an empty set, that component is discarded. Finally, the block index is employed to retrieve the blocks that have been retained in the component list, which are the only blocks considered by BrewER for the processing.

Ordering Entities while Resolving Them

The desired entity resolution on-demand algorithm is an iterative algorithm capable of identifying the next entity that should be cleaned and emitted as soon as possible, i.e., with the fewest calls to the matching function μ^Q. This can be approximated by determining the lower/upper bound of the value of the ORDER BY clause for the entity, so to define whether it should be emitted before or after all other entities.

In fact, each entity has as ordering value the highest or lowest value of its records (recall the definition of record bounds presented in Section 2.4.1). For instance, consider the first entity that has to be emitted: its final value might be determined by one or more records that are not in the seed set and that are higher/lower than all values of the seeds. Hence, the comparisons cannot involve only records in the seed set, but a broader set, composed of both the seeds and their candidate matches, must be considered. Each element of that set can be inserted in a priority queue (according to the value of each record); then, the algorithm can iteratively evaluate the head (i.e., the record with the current highest/lowest value) and determine its bound. Thus, as soon as a record has a lower/upper bound that is greater/lower than or equal to the head of the queue, it can be emitted.

*Figure 2.8a shows a dirty dataset that Ellen (the data scientist) wants to query with the query Q_1^c of Figure 2.7. Figures 2.8b and 2.8c report the results of the query employing AVG and MIN aggregate functions on **`price`**, respectively. Ellen is employing AVG(**`price`**) and a blocking strategy that inserts in the same block all the records that share at least one token in **`brand`** (the blocks are at the top of Figure 2.9).*

*With a traditional entity resolution approach, 12 pairs of records are compared (6 pairs from the block "*canon*" and 6 from the block "*nikon*") to clean the dataset and obtain the first results for the query Q_1^c. Hence, Ellen has to wait the time for the complete entity resolution for even just the first correct result, namely ε_1. Instead, if she employs* BrewER*, ε_1 is returned after just 5 comparisons, as explained next.*

First, the seed query of Q_1^c is generated (Figure 2.7c). The seed records $\{r_1, r_3, r_5, r_7\}$ are then extracted with Q_1^{seed} and processed by BrewER *with*

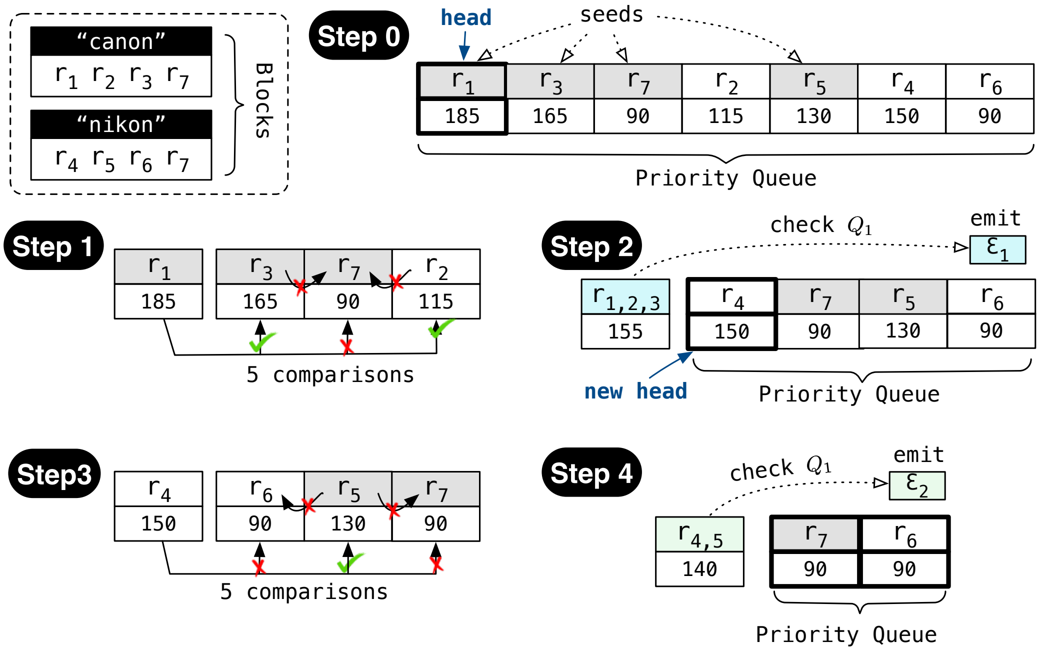


Figure 2.9: Example of BREWER functioning in the AVG/DESC case.

their candidates r_2, r_4 and r_6 (from the blocks). The process is depicted step by step in Figure 2.9.

A priority queue is populated with the seeds and their candidates, according to their ordering value (Step 0). We employ a max priority queue because the required ordering is descending, otherwise it should be a min priority queue. The head of the queue potentially belongs to the entity with the highest price, i.e., the first entity to be emitted.

All the records that match with the head have to be identified to compute the average of the ordering values. Thus, as shown in Step 1, the head record is compared to all its candidates. For each matching record, BREWER *recursively compares also its candidates (if not already compared), so to obtain a final entity cluster, as the matching function might not be transitive.*

At Step 2, the entity ε_1 to which the head r_1 belongs can be bounded, i.e., its ordering value is now known. At this stage, if the value of ε_1 is greater or equal than the new head record value of the queue (i.e., r_4), ε_1 can be emitted, since no other entity can have a greater ordering value. Yet, we need to check whether ε_1 actually satisfies Q_1^c. Hence, the filtering predicates of Q_1 are applied. A total of 5 comparisons have been executed to provide Ellen the first correct result for her query, in its correct order. Then, in Steps 3 and 4 the same process is repeated.

2.5.2 The BrewER Algorithm

The BrewER algorithm can handle any combination of *bounded aggregations* (see Section 2.4.1) and ordering (ASC/DESC). Here, for ease of presentation, we consider the case of entities emitted in non-increasing order of a given attribute, i.e., ORDER BY $\alpha(\cdot)$ DESC, and $\alpha(\cdot) \equiv \text{MAX}(\cdot)$ as resolution function of the ordering value. With other aggregate functions (AVG, VOTE, etc.) the presented algorithm does not change. Also, when we refer to the *ordering value* (or simply *value*) of a record or of an entity, we mean the value assumed by the ORDER BY attribute. These are the only values of the records that are relevant to the algorithm, i.e., the values that can affect the order of emission of the entities.

Algorithm Description

As input, the BrewER algorithm (Algorithm 2.1) takes the dirty dataset $\mathcal{D}$, a query Q^c, and the lists *CandLists*, *MatchLists* and *NonMatchLists*. *CandLists* is a list whose i-th element is a list itself containing all candidate matches of the i-th record in $\mathcal{D}$ generated with blocking. If blocking is not employed, BrewER considers the all-pairs comparison scenario. With large datasets, it is always preferable to employ blocking to avoid the quadratic complexity. Thus, we assume that *CandLists* fits in memory (as it does for all the experiments in Section 2.7). Alongside *CandLists*, the two complementary lists of lists *MatchLists* and *NonMatchLists* keep track of the matches and non-matches that have already been compared with μ^Q, respectively. They have the same size of *CandLists*, but they can be implemented with lists of bit arrays, thus accounting for a low memory overhead. We use the following notation: *MatchLists*[i][j] (or *NonMatchLists*[i][j]) is the flag indicating the match (non-match) between r_i and its candidate matching record r_j. Each element in the lists is initialized to zero, i.e., *false*. *MatchLists* and *NonMatchLists* keep track of matching/non-matching pairs among multiple query executions, avoiding redundant comparisons. As output, Algorithm 2.1 emits the resolved entities incrementally, according to the ORDER BY clause of Q^c. In the following, we describe the details of the algorithm.

First, the seed query Q^{seed} is derived from Q^c (Line 1), as defined in Section 2.5.1, and issued on $\mathcal{D}$ to obtain the seed records and to initialize the set of seed identifiers *Seeds* (Line 2). This set is then merged with all candidate matches of the seeds in Line 3. *MatchSet* (Line 4) is an empty set used to keep track of the records that have already been positively matched. *EntityMap* (Line 5) is a *map* data structure (e.g., a *hash table*) that stores *key-value* pairs: the *key* is the *id* of a representative record of an entity, and

Algorithm 2.1: BrewER algorithm with $\alpha(\cdot) \equiv$ MAX$(\cdot)$ and DESC ordering.

```
Input: $\mathcal{D}$; $Q^c$; $CandLists$; $MatchLists$; $NonMatchLists$.
Output: The incremental emission of the resolved entities.
1  $Q^{seed} \leftarrow$ get the seed query for $Q^c$
2  $Seeds \leftarrow \Pi_{id}\, Q^{seed}(\mathcal{D})$                      // seed record ids
3  $\mathcal{I} \leftarrow Seeds \bigcup \{j \in CandLists[i] \mid i \in Seeds\}$
4  $MatchSet \leftarrow \emptyset$                                     // empty set
5  $EntityMap \leftarrow Map(\emptyset)$                          // empty hash table
6  $PQueue \leftarrow maxHeap(\emptyset)$                           // priority queue
7  forall $i \in \mathcal{I}$ do
8      $r_i \leftarrow \sigma_{id=i}(\mathcal{D})$
9      $val \leftarrow r_i[Q^c.orderByAttribute]$
10     $PQueue.insert(i, val)$
11 while $PQueue \neq \emptyset$ do
12     $i \leftarrow PQueue.extractHeadElement()$
13     if $EntityMap.hasKey(i)$ then
14         emit $EntityMap.get(i)$
15         continue                                          // go to Line 11
16     if $i \in MatchSet$ then
17         continue                                          // go to Line 11
18     $E \leftarrow \emptyset$                      // initialize the entity cluster set
19     $\mathcal{R} \leftarrow \emptyset$                       // initialize the records to check
20     $onlySeeds \leftarrow True$                        // consider seeds only
21     $recordID \leftarrow i$
22     matchingProcedure()
23     if $E \equiv \emptyset$ and $i \notin Seeds$ then
24         continue                  // no matching seeds: go to Line 11
25     $E \leftarrow E \bigcup \{i\}$
26     $onlySeeds \leftarrow False$                     // consider also non-seeds
27     while $\mathcal{R} \neq \emptyset$ do
28         $recordID \leftarrow$ extract an id from $\mathcal{R}$
29         matchingProcedure()
30     $\varepsilon_i \leftarrow \widetilde{Q}^c(\sigma_{id \in E}(\mathcal{D}))$                   // a resolved entity
31     if $\varepsilon_i \neq \emptyset$ then
32         $EntityMap.add(i, \varepsilon_i)$
33         $PQueue.insert(i, \varepsilon_i.val)$
```

Algorithm 2.2: matchingProcedure

```
Input: Procedure 2.2 has visibility of all variables in Algorithm 2.1.
1  $Candidates \leftarrow CandLists[recordID]$
2  for $(p = 0; p{+}{+}; p < len(Candidates))$ do
       /* $p$ is the position of the current candidate in the
          candidate list of recordID                              */
3      $candidateID \leftarrow Candidates[\mathrm{p}]$
4      if $onlySeeds == True \wedge candidateID \notin Seeds$ then
5          break                          // continue only for non-seeds
6      else if $candidateID \in E$ then
7          continue                                   // go to Line 2
8      else if $MatchLists[recordID][p]$ then
9          $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{candidateID\}$
10         $E \leftarrow E \bigcup \{candidateID\}$
11     else if $NonMatchLists[recordID][p]$ then
12         continue                                   // go to Line 2
13     else if $\mu^Q(\mathcal{D}[i], \mathcal{D}[candidateID])$ then
14         $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{candidateID\}$
15         $E \leftarrow E \bigcup \{candidateID\}$
16         $MatchLists[recordID][p] \leftarrow True$
17         $p' \leftarrow$ get position of recordID in $CandLists[candidateID]$
18         $MatchLists[candidateID][p'] \leftarrow True$
19     else
20         $NonMatchLists[recordID][p] \leftarrow True$
21         $p' \leftarrow$ get position of recordID in $CandLists[candidateID]$
22         $NonMatchLists[candidateID][p'] \leftarrow True$
23 $MatchSet \leftarrow MatchSet \bigcup \{recordID\}$
```

the value is the resolved entity satisfying Q^c. In practice, *EntityMap* stores the entities that have been resolved and are therefore ready to be emitted when their turn comes.

A *max heap* (or *min heap* in the case of ASC ordering) is initialized in Line 6 and populated with the seeds and their candidates, serving as *priority queue* (Lines 7-10). The basic idea is to iteratively extract the head element from the priority queue, resolve its corresponding entity ε, and insert ε into the priority queue with its consolidated ordering value. Thus, every time that the head of the priority queue is a resolved entity, it can be emitted: all other records and entities in the queue have an equal or lower value.

The iteration on the priority queue starts in Line 11. Notice that the priority queue stores only record *id*s associated to their ordering values. The head element i of the priority queue is extracted in Line 12 and then:

- (Line 13) If i is the representative *id* of an entity that was completely resolved in a previous iteration, that entity has the highest value of all possible remaining entities/records in the priority queue. Thus, it is emitted in Line 14.
- (Line 16) If i is not a representative record, but has already been matched in a previous iteration with at least one other record, the iteration continues and the next element in the priority queue is considered (Line 17).
- (Lines 18-33) Otherwise, the record r_i corresponding to i is compared to its candidates to completely resolve its entity or discarded, as explained in the following.

An entity should be emitted only if it is derived from at least one seed record (otherwise it does not satisfy Q^c); so, the first comparisons to be performed are those to ensure that r_i matches a seed record (if it is not a seed record itself). This is checked in Lines 20-24. If r_i matches a seed (or it is a seed itself), then all remaining candidates of r_i are considered, and the entity ε_i is completely resolved (Lines 26-30). BrewER tries to find also the matches of each match recursively, starting from r_i. To do so, it recursively inserts the *id*s of matches in $\mathcal{R}$.

The actual comparisons are verified by calling the matchingProcedure (Procedure 2.2), which also updates $\mathcal{R}$ with newly discovered matches. In the first call to the matchingProcedure (Line 22), the flag *onlySeeds* is set to true, so to check only seed records. The set E collects the *id*s of the matches of r_i. After the first call to Procedure 2.2, if E is empty, then no seed matches r_i and the execution is interrupted (Line 24). The other calls to the matchingProcedure have the *onlySeeds* flag equal to false instead.

So, at the end of the while loop on $\mathcal{R}$ of Line 27 of Algorithm 2.1, all matching records of r_i are in E. The resolution functions can now be applied to that cluster of records. To do so, Algorithm 2.1 issues the query $\widetilde{Q}^c$ on the set of identified matching records, i.e., $\sigma_{id \in E}(\mathcal{D})$ (Line 30). $\widetilde{Q}^c$ denotes the query Q^c without the matching function (μ^Q) invocations: at this stage of the algorithm the query is performed against a cluster of known matching records E, i.e., it can assume that the GROUP BY ENTITY yields only one group. Thus, by issuing $\widetilde{Q}^c$ on the set of matching records, all aggregations are applied and all clauses of the query are verified in order to yield a single representative record ε_i. Depending on the clauses of $\widetilde{Q}^c$, ε_i can also be an empty set.

Finally, if not an empty set, ε_i is added to the *map* data structure as a value for the key i, and i is added back to the priority queue associated with the value of ε_i. The loop ends when the priority queue is empty, i.e., when all entities satisfying the query have been emitted.

We now describe the matchingProcedure (Procedure 2.2) in detail. Given a record (*recordID*), it seeks for its matches iterating among its candidates. Lines 4-5 are needed to manage the first calls of Algorithm 2.1 mentioned above, which compare only the seed records (note that the following calls do not need this check). Firstly, for each candidate, the matchingProcedure checks if it has already been assigned to the current entity cluster E (Line 6). This may happen when "following" the match: we do not want to execute a comparison with a record already assigned to the entity cluster of *recordID*. For example, in Figure 2.9, we want to avoid comparing r_2 and r_3 and vice versa, since they already are in the entity cluster of r_1. Then, the matching-Procedure checks whether the candidate pairs involving r_i have already been compared, in Lines 8 and 11, by exploiting *MatchLists* and *NonMatchLists*. If both *MatchLists*[*recordID*][p] and *NonMatchLists*[*recordID*][p] are equal to zero (i.e., *false*), this means that the comparison has not been executed yet. Hence, the matchingProcedure invokes the matching function in Line 13 (denoted with the notation μ^Q). If the candidate record is a match, then it is inserted into $\mathcal{R}$ (Line 14). Finally, the candidate is added to E to avoid checking it again in further calls (Line 15), and the current *recordID* to the *MatchSet* (Line 23).

Special Case: Discordant Ordering Queries

We present a variation of the BrewER algorithm called *Discordant* BrewER, which introduces an optimization for a special yet frequent case of queries, namely queries that order the entities with the following predicates: *(i)* ORDER BY MIN(·) DESC and *(ii)* ORDER BY MAX(·) ASC (here only the former case is discussed; it is trivial to adapt for the latter). We call this case *discordant ordering* because the first entity to be emitted is the one with the maximum value, which is in turn the minimum value among the records that compose that entity.

Discordant BrewER is based on the observation that if a seed record matches with a non-seed record with a higher value, the value of the latter is not for certain the value of the final entity. Thus, the values of seed records belonging to an entity ε determine the maximum value that ε can assume: non-seed records can only lower that value, if they match. Hence, the heap in Algorithm 2.1 can be initialized by considering only seed records, i.e., by omitting the union with the candidates in Line 3. This significantly reduces the searching space and allows to achieve correct results with a fraction of the comparisons needed by the general algorithm, as we show with the experiments of Section 2.7.3.

2.6 Use Case Examples

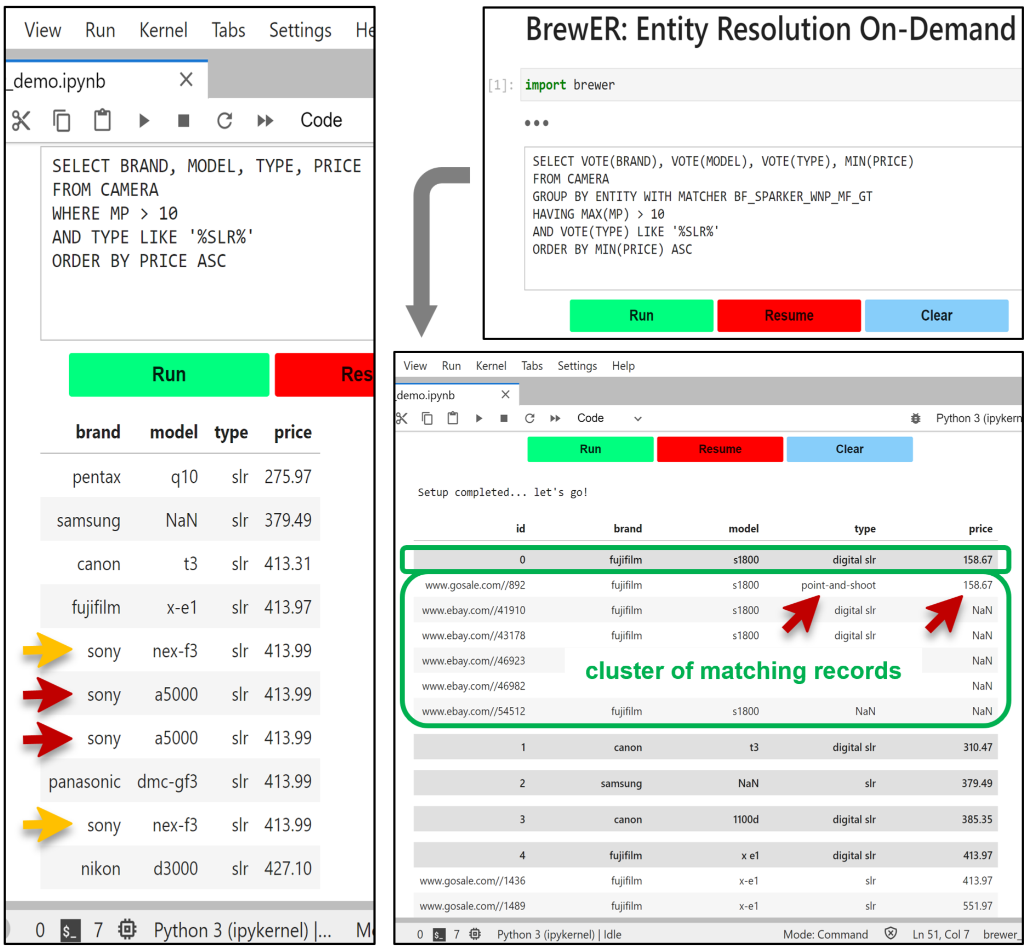
This section reports two relevant use case examples, presented in our demonstration [Zecchini et al., 2023], which show how data practitioners can benefit from BrewER for their real-world tasks, especially from its seamless integration in Python workflows in Jupyter notebooks (where it can be easily combined with blocking and matching functions from their favorite libraries). In particular, the two use cases aim to show through example stories: *(i)* how BrewER can by used by practitioners to run queries on dirty data, e.g., for quick data exploration (Section 2.6.1); *(ii)* how BrewER can be used by practitioners for debugging their entity resolution pipelines (Section 2.6.2).

2.6.1 Querying Dirty Datasets

Mary is a data analyst who needs to build a BI dashboard to analyze the price and the characteristics of cameras sold on several popular e-commerce stores. She has been asked to consider only SLR cameras, with a minimum resolution of 10 MP, and to focus on those with the lowest price among them. Further, she has been asked to extract the data with short notice and with a strict deadline for a company business meeting.

Fortunately, it is very simple for Mary, who knows SQL, to come up with a query to find the cheapest SLR cameras with at least 10 MP in resolution. Unfortunately, by issuing the SQL query on the dirty data, she obtains inconsistent results. In fact, considering the results depicted in Figure 2.10a, she notices the presence of duplicate records (e.g., the two records describing the Sony A5000 camera) and other data quality issues.

Mary already has some pre-trained matchers from previous projects on dirty product datasets that she can try on this new data, or she can exploit


View
Run
Kernel
Tabs
Settings
demo.ipynb
Code
Run
Res
0
7
Python 3 (ipykernel) |...
BrewER: Entity Resolution On-Demand
Resume
Clear
Help
Python 3 (ipykern
Python 3 (ipykernel) | Idle
Mode: Command
Ln 51, Col 7
brewer_
cluster of matching records

```
SELECT BRAND, MODEL, TYPE, PRICE
FROM CAMERA
WHERE MP > 10
AND TYPE LIKE '%SLR%'
ORDER BY PRICE ASC
```

brand	model	type	price
pentax	q10	slr	275.97
samsung	NaN	slr	379.49
canon	t3	slr	413.31
fujifilm	x-e1	slr	413.97
sony	nex-f3	slr	413.99
sony	a5000	slr	413.99
sony	a5000	slr	413.99
panasonic	dmc-gf3	slr	413.99
sony	nex-f3	slr	413.99
nikon	d3000	slr	427.10

```
[1]: import brewer
```

```
SELECT VOTE(BRAND), VOTE(MODEL), VOTE(TYPE), MIN(PRICE)
FROM CAMERA
GROUP BY ENTITY WITH MATCHER BF_SPARKER_WNP_MF_GT
HAVING MAX(MP) > 10
AND VOTE(TYPE) LIKE '%SLR%'
ORDER BY MIN(PRICE) ASC
```

```
Setup completed... let's go!
```

id	brand	model	type	price
0	fujifilm	s1800	digital slr	158.67
www.gosale.com//892	fujifilm	s1800	point-and-shoot	158.67
www.ebay.com//41910	fujifilm	s1800	digital slr	NaN
www.ebay.com//43178	fujifilm	s1800	digital slr	NaN
www.ebay.com//46923				NaN
www.ebay.com//46982				NaN
www.ebay.com//54512	fujifilm	s1800	NaN	NaN
1	canon	t3	digital slr	310.47
2	samsung	NaN	slr	379.49
3	canon	1100d	digital slr	385.35
4	fujifilm	x e1	digital slr	413.97
www.gosale.com//1436	fujifilm	x-e1	slr	413.97
www.gosale.com//1489	fujifilm	x-e1	slr	551.97

(a) SQL on dirty data (b) SQL with BrewER on dirty data

Figure 2.10: Querying dirty datasets with BrewER.

a pay-as-you-go LLM-based service (e.g., GPT-3) as a binary matcher. She also knows that she can employ some unsupervised blocking techniques to accelerate the entity resolution process, but it would still take hours and a significant amount of resources to clean the entire dataset.

BrewER *allows Mary to overcome this situation and quickly obtain a consistent result. As depicted in Figure 2.10b, she can write the query in a dedicated text area of the notebook, generated using a simple Jupyter widget. Within the query, she declares the matcher to be used in the specific GROUP BY ENTITY clause and the aggregation functions for the attributes of interest (e.g., the minimum value for the price). Then, after clicking on the green* Run *button, the resulting cleaned entities will start to appear in the area below* *as soon as they are obtained, one by one in a progressive fashion, correctly sorted by ascending price. The execution will automatically stop after the emission of the number of entities required by Mary.*

As we can see in Figure 2.10b, the top cleaned entities are significantly different from the results obtained on the dirty data shown in Figure 2.10a. BrewER *allows Mary to analyze the produced entities through the UI, to better understand what happened during the entity resolution process. In fact, by simply clicking on the row representing the entity, she can expand the table and visualize the matching records that were aggregated to produce it, understanding why an attribute presents a certain value. Through this feature, Mary can discover the reasons behind the inconsistencies of the dirty results: for instance, the record determining the price of the cheapest model did not fulfil the condition defined on the type in the WHERE clause, thus was erroneously discarded.*

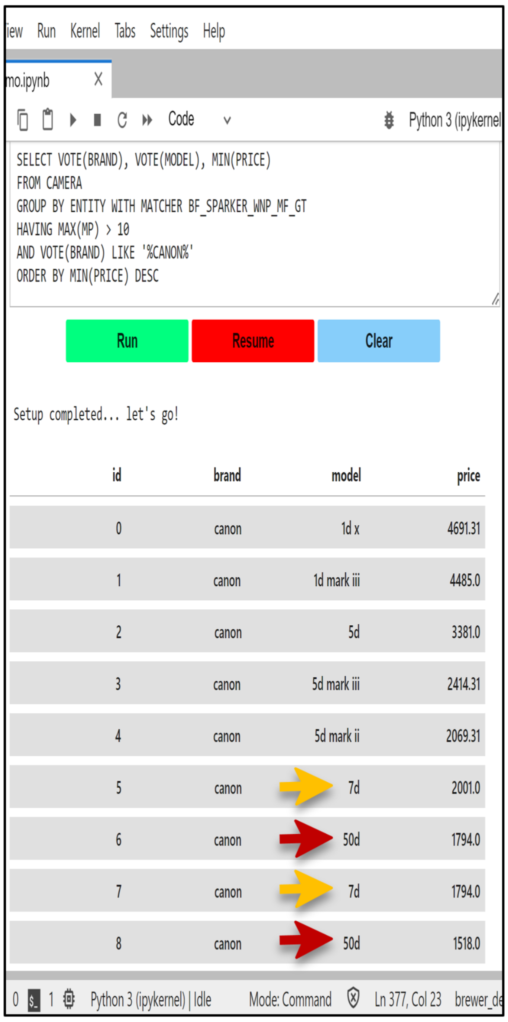
2.6.2 Entity Resolution Pipeline Debugging

BrewER *can significantly speed up not only the access to the cleaned results of queries issued on dirty datasets, but also the design of entity resolution pipelines. As shown in the previous section, the choice of different aggregation functions, as well as different combinations of blocking and matching functions, can indeed determine significantly different results.*

Anna is a data engineer that employs BrewER *to get quick insights into the goodness of the entity resolution pipeline she is designing at a negligible cost compared to existing solutions. In fact,* BrewER *automatically and dynamically selects a portion of the data to be cleaned relevant for the task at hand, expressed through a query.*

By assessing the quality of the entity resolution pipeline on that portion, Anna is more confident that it will be suited for the task compared to a pipeline tuned on a random sample of the data. Further, by having access to the entities in the result progressively as soon as they are produced, Anna can quickly interrupt the entity resolution process as soon as she spots an issue, saving a significant amount of time and resources. In fact, with a batch approach she would have to wait until the end of the processing of the entire batch of data to spot the same issue.

BrewER *perfectly supports exploratory top-*k *queries, allowing to debug the designed entity resolution pipeline during the cleaning process in a stop-and-resume fashion. For instance, if a top-10 query produces the results in Figure 2.11a, Anna understands from the presence of duplicates the inaccuracy of her pipeline and she can immediately stop the cleaning process to solve the problem. The designed blocking technique was too aggressive and pruned*



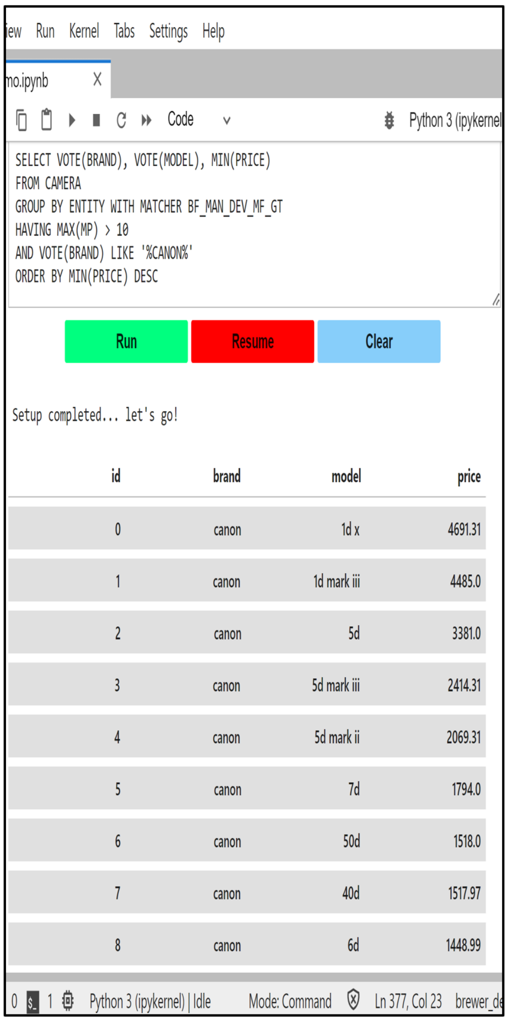
```
SELECT VOTE(BRAND), VOTE(MODEL), MIN(PRICE)
FROM CAMERA
GROUP BY ENTITY WITH MATCHER BF_SPARKER_WNP_MF_GT
HAVING MAX(MP) > 10
AND VOTE(BRAND) LIKE '%CANON%'
ORDER BY MIN(PRICE) DESC
```

Setup completed... let's go!

id	brand	model	price
0	canon	1d x	4691.31
1	canon	1d mark iii	4485.0
2	canon	5d	3381.0
3	canon	5d mark iii	2414.31
4	canon	5d mark ii	2069.31
5	canon	7d	2001.0
6	canon	50d	1794.0
7	canon	7d	1794.0
8	canon	50d	1518.0

iew Run Kernel Tabs Settings Help
mo.ipynb X
Code
Python 3 (ipykernel
Run
Resume
Clear
0 1 Python 3 (ipykernel) | Idle Mode: Command Ln 377, Col 23 brewer_de

(a) BrewER with blocker1



```
SELECT VOTE(BRAND), VOTE(MODEL), MIN(PRICE)
FROM CAMERA
GROUP BY ENTITY WITH MATCHER BF_MAN_DEV_MF_GT
HAVING MAX(MP) > 10
AND VOTE(BRAND) LIKE '%CANON%'
ORDER BY MIN(PRICE) DESC
```

Setup completed... let's go!

id	brand	model	price
0	canon	1d x	4691.31
1	canon	1d mark iii	4485.0
2	canon	5d	3381.0
3	canon	5d mark iii	2414.31
4	canon	5d mark ii	2069.31
5	canon	7d	1794.0
6	canon	50d	1518.0
7	canon	40d	1517.97
8	canon	6d	1448.99

iew Run Kernel Tabs Settings Help
mo.ipynb X
Code
Python 3 (ipykernel
Run
Resume
Clear
0 1 Python 3 (ipykernel) | Idle Mode: Command Ln 377, Col 23 brewer_de

(b) BrewER with blocker2

Figure 2.11: Entity resolution pipeline debugging with BrewER.

some matches from the candidate set, preventing the complete resolution of some entities. After changing the blocking settings, Anna can run her query again: Figure 2.11b shows how the new setting solved the issue.

Since BrewER *allows saving the status of the cleaning process in case of early termination, this time Anna can simply click on the* Resume *button to continue the cleaning process, running for instance a further top-*k *query for inspecting more entities, then resuming the process again for the complete emission of the results.*

2.7 Experimental Evaluation

This section aims to answer the following questions:

Q1. What is the performance of BrewER *when executing entity resolution on-demand?* (Section 2.7.1)

Q2. How do entity resolution on-demand baselines derived from traditional batch and progressive entity resolution methods perform? (Section 2.7.2)

Q3. What is the improvement in the case of discordant ordering queries, introduced at the end of Section 2.5.2? (Section 2.7.3)

Q4. How well does BrewER *perform with different aggregate functions?* (Section 2.7.4)

Q5. How does BrewER *perform with blocking?* (Section 2.7.5)

Q6. How does BrewER *perform with missing values?* (Section 2.7.6)

Q7. How fast is BrewER *and how much time does it save without cleaning the entire dataset?* (Section 2.7.7)

Datasets

We employ four real-world datasets from multiple domains with different sizes and characteristics, summarized in Table 2.1. For all of them the ground truth is known. The first dataset, SIGMOD20 [Crescenzi et al., 2021; Zecchini et al., 2020], is composed of camera specifications extracted from 24 e-commerce websites and has been employed for the SIGMOD 2020 Programming Contest[3]. The second dataset is SIGMOD21, provided by Altosight[4] for the SIGMOD 2021 Programming Contest[5], which contains well-curated specifications of electronic products (mainly USB pen drives) scraped from more than 20 websites. The third dataset, Altosight, is a superset of SIGMOD21, but differently from it, this larger set of entities is not well-curated and presents many noisy records with redundant values, missing values, and/or HTML tags. The last dataset is Funding[6], which reports financing requests addressed to the NYC Council Discretionary Funding. Entity resolution can

[3]`http://www.inf.uniroma3.it/db/sigmod2020contest/`

[4]`https://altosight.com/`

[5]`https://dbgroup.ing.unimo.it/sigmod21contest/`

[6]`https://raw.githubusercontent.com/qcri/data_civilizer_system/master/grecord_service/gr/data/address/address.csv`

Table 2.1: Characteristics of the selected datasets. For each dataset are reported the number of records (#D), the number of matches (#M), the number of entities (#E) with their average size (AVG Size), the number of attributes (#A), and finally the ordering attribute (OA).

Dataset	**#D**	**#M**	**#E (AVG Size)**	**#A**	**OA**
SIGMOD20	13.58k	12.01k	3.06k (4.4)	4	`megapixels`
SIGMOD21	1.12k	1.08k	190 (5.9)	4	`price`
Altosight	12.47k	12.44k	453 (27.534)	4	`price`
Funding	17.46k	16.70k	3.11k (5.6)	17	`amount`

be performed on it to identify the organizations presenting these requests, as done by Deng et al. [D. Deng et al., 2019].

We preprocess all datasets by casting the ordering values to floats, lowercasing all attributes, and filtering out records with a null value in the ordering attribute. The null values do not affect the final ordering of the entities (i.e., they are not considered by the aggregate functions), but slow down the computation for those entities that present a lot of them.

Experimental Setup

BrewER has been implemented in Python 3.7. Our experiments were performed on a server machine equipped with an Intel Xeon Silver 4116 CPU @ 2.10 GHz, a Nvidia Tesla T4 GPU, and 100 GB of RAM.

2.7.1 Performance of the BrewER Algorithm

Here, we want to assess how BrewER performs in terms of comparisons required to progressively return the result sets of queries. As a baseline we employ QDA [Altwaijry et al., 2013], which applies conflict resolution function directly to each comparison; thus, it can work only with MIN and MAX aggregate functions. For this reason, we consider only these two aggregate functions in this section. The following Section 2.7.4 shows the performance of BrewER with other aggregate functions.

Since the goal of this experiment is to evaluate the BrewER algorithm (i.e., Algorithm 2.1), we do not employ any blocking strategy, which would influence the overall performance. We study how BrewER performs with blocking in Section 2.7.5. Finally, we are not interested in neither designing nor discovering the best matching functions for the task; hence, as a matcher, we employ an oracle that correctly labels all the comparisons (remember that the ground truth is known and BrewER is matcher-agnostic).

Table 2.2: Minimum, maximum, and average cardinality of the result sets of the considered batches of queries.

	Conjunctive Queries			**Disjunctive Queries**		
Dataset	**#MIN**	**#MAX**	**#AVG**	**#MIN**	**#MAX**	**#AVG**
SIGMOD20	27	172	55.63	368	567	440.55
SIGMOD21	5	15	7.43	28	85	55.45
Altosight	9	32	18.40	87	193	139.08
Funding	8	212	42.13	336	2297	1259.05

Query Generation

We generate two basic types of synthetic queries for our experiments. Firstly, *conjunctive queries*, i.e., queries with two selection predicates employing the LIKE operator in AND to express queries on related attributes (e.g., to select the prices of a series of specific models produced by a brand). Secondly, *disjunctive queries*, i.e., queries with two selection predicates in OR referring to the same attribute (e.g., to select all models produced by two brands).

The column OA of Table 2.1 indicates the ordering attribute employed for each dataset. For the AND queries on SIGMOD20, the selection is based on a random value among the ten most frequent brands and on a random model series associated with the selected brand; for the OR queries, on two random values chosen from the ten most frequent brands. For the AND queries on SIGMOD21 and Altosight, the selection is based on the list of all nine brands in the dataset and a list of the most common USB stick sizes; for the OR queries, on two random brands. Finally, for the AND queries on Funding, the selection is based on a list of the most common areas in which the organizations are active and on a frequent substring of their names (e.g., *association*, *inc*, etc.); for the OR queries, on two random areas.

For each dataset, we consider two batches of 20 queries: one for the conjunctive and one for the disjunctive case. Since the goal is to study the progressive emission of the resulting entities, each set is composed of the 20 queries emitting the highest number of entities out of a set of at least 50 randomly generated queries. Their characteristics are described in Table 2.2.

Measures

For each batch of conjunctive/disjunctive queries executed on a certain dataset, we compute the *progressive average macro-recall* (*progressive recall* for simplicity), as follows. The recall for a query Q_i is defined as:

$$recall_{Q_i} = \frac{\#\{emitted\ entities\}}{\#Q_i^c(\mathcal{D})}$$

where $\#Q_i^c(\mathcal{D})$ is the cardinality of the result set for the query Q_i. For each query Q_i in a batch of 20, we track the recall by steps of 5% of the total number of comparisons entailed by Q_i (i.e., a total of 20 steps). Thus, 20 values of $recall_{Q_i}$ are collected for each Q_i in the batch. For instance, if Q_1 entails one million of comparisons, we record the recall of its execution in BrewER every 50,000 comparisons (5% of one million). Then, to obtain aggregate values for each batch of queries:

(i) we compute the average number of comparisons for each step among the queries:

$$avg.\ num.\ comp. = \frac{\sum_{Q_i \in [Q_1,..,Q_N]} \#executed\ comparisons\ for\ Q_i}{N}$$

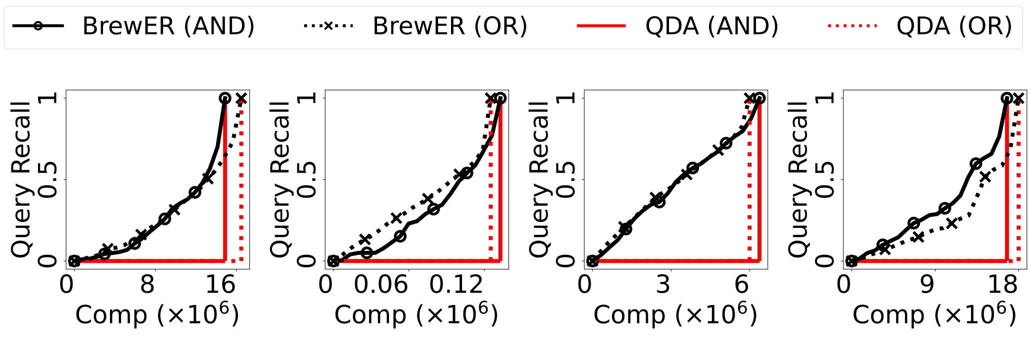
(ii) we compute the average value of recall corresponding at that step, i.e., the *macro-recall* for a batch of queries (or simply *query recall*):

$$query\ recall = \frac{\sum_{Q_i \in [Q_1,..,Q_N]} recall_{Q_i}}{N}$$

In our experiments, $N = 20$; thus, for each batch of queries, the progressive recall is represented by 20 points (one for each step) that can be reported in a single plot to summarize the performance of an entity resolution algorithm on that batch.

Baseline

We adapted QDA [Altwaijry et al., 2013] to process queries that contain predicates on categorical attributes. QDA does not provide any mechanism to handle ORDER BY clauses, thus the result of its execution is a *batch* version of each query, i.e., performing the sorting of the entities at the end of the resolution process. Also, it supports only MIN and MAX as aggregate functions, since it merges (i.e., resolves) pairs of records as soon as they are found to be matching (this is not compatible with aggregate functions like VOTE and AVG, which take as input more than two values). In a nutshell, QDA tries to discard the entities that are not part of the result as soon as possible, by incrementally matching pairs of records that belong to those entities. In practice, by using our terminology, QDA tries to match all the seed records first, as in our Algorithm 2.1 we do by checking the match with the seed records. Hence, BrewER and QDA perform the same number of comparisons if enough time is given.



(a) SIGMOD20 (b) SIGMOD21 (c) Altosight (d) Funding

Figure 2.12: Progressive recall with BrewER.

Results

Figure 2.12 shows the average progressive recall obtained through the execution of the described randomly generated batches of queries, for each dataset and type of query.

QDA shows a typical step curve for this task due to the fact that it has to compare all candidate pairs before starting emitting results. On the other hand, BrewER exhibits a progressively increasing recall for all four datasets as a function of the executed comparisons. We do not observe particular differences in performance among the datasets. Also, the kind of query (AND/OR) does not affect the performance. On SIGMOD20 (Figure 2.12a) and Funding (Figure 2.12d), disjunctive queries entail a higher number of comparisons than the conjunctive queries, on average (at most 15% more); vice versa, for SIGMOD21 (Figure 2.12b) and Altosight (Figure 2.12c), conjunctive queries need more comparisons (at most 10% more).

This is due to the generation of the seed records: as explained in Section 2.5.1, the seed records of a query are extracted with a disjunctive query and employed in Algorithm 2.1. Thus, the final number of comparisons depends on the selectivity of each predicate, and not on the result size.

2.7.2 Shortcomings of the Traditional Baselines

We compare BrewER against two baselines that we derived from: *(i)* a traditional batch entity resolution workflow; *(ii)* an existing progressive entity resolution method. We call the former *Batch-Query-Baseline* and the latter *Progressive-Query-Baseline*. Our goal is to show that adapting exist-

Table 2.3: BrewER vs. Batch-Query-Baseline.

	BrewER		**Batch-Query-Baseline**					
Dataset	R, P, F_1	Err@x	R	P	F_1	Err@1	Err@5	Err@20
SIGMOD20	1.00	0%	0.89	0.99	0.92	30%	13%	9%
SIGMOD21	1.00	0%	0.91	0.50	0.60	30%	40%	42%
Altosight	1.00	0%	0.89	0.20	0.31	60%	45%	57%
Funding	1.00	0%	0.71	0.86	0.77	100%	50%	70%

ing entity resolution methods to produce *correct* results for a query without cleaning the entire data or without designing a specific progressive algorithm is not trivial.

Batch-Query-Baseline

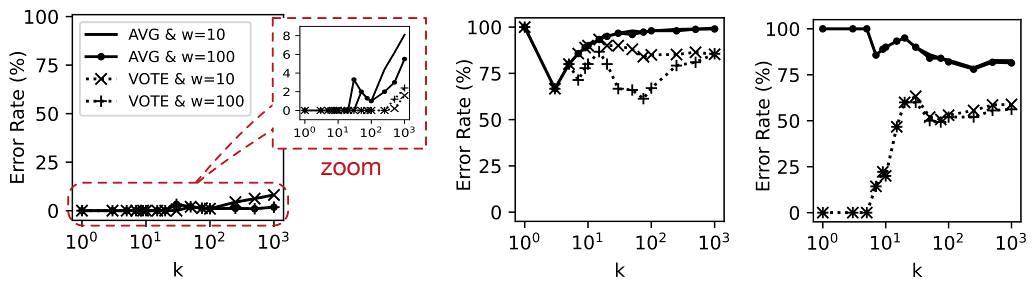
Algorithm 2.1 guarantees that the results of a query issued on top of a dirty dataset are emitted as if the query was issued on the cleaned version of the dataset, i.e., $Q^c(\mathcal{D}) \equiv Q(\mathcal{D}^c)$. Yet, how good would the results be if we simply issue the query directly on the dirty dataset (i.e., $Q(\mathcal{D})$) and then perform entity resolution on just that portion of the data?

This question arises from observing that $Q(\mathcal{D}) \not\equiv Q(\mathcal{D}^c)$. In fact, by issuing the query Q (e.g., Q_1 in Figure 2.7a) directly on the dirty data, relevant records might be filtered out (e.g., records r_2 and r_4 in the example of Figure 2.8a).

To investigate this effect, Batch-Query-Baseline first filters the dirty data $\mathcal{D}$ with Q and then performs entity resolution on the result $Q(\mathcal{D})$. We compare it against BrewER by executing a batch of ten randomly generated AND queries for each dataset. For each query q, we consider the set of matching pairs $\mathcal{M}_q$ needed to identify the entity set that satisfies q (we know $\mathcal{M}_q$ from the ground truth of each dataset) and the set of matches $\mathcal{M}_\varepsilon$ that the considered method identifies for producing the results. Then, we compute recall R_q, precision P_q, and F_1-score F_{1q} as follows:

$$R_q = \frac{|\mathcal{M}_q \bigcap \mathcal{M}_\varepsilon|}{|\mathcal{M}_q|} \qquad P_q = \frac{|\mathcal{M}_q \bigcap \mathcal{M}_\varepsilon|}{|\mathcal{M}_\varepsilon|} \qquad F_{1q} = \frac{2 \cdot R_q \cdot P_q}{R_q + P_q}$$

The results of the comparison are shown in Table 2.3, where we report the average of recall, precision, and F_1-score for each batch of queries. BrewER always returns the correct answer, and thus recall, precision, and F_1-score are always 1.00. We also report the *error rate* (*Err@k*) of a method, which is the percentage of erroneously yielded entities in the *first* k emitted entities.



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 2.13: Progressive-Query-Baseline error rate.

For instance, $Err@20 = 42\%$ means that among the first 20 entities emitted according to the ORDER BY clause, 42% are incorrect according to the ground truth. These errors are introduced by filtering the dirty data with Q. For example, applying Q_1 directly to the dirty dataset of Figure 2.8a and considering AVG as resolution function for `price`, ε_1 ends up with a price of \$175 (since r_2 is filtered out), instead of \$155, which is ε_1 price value in the ground truth (Figure 2.8b).

Table 2.3 shows how the error rate is significant for all the datasets and for different values of k. On the other hand, BrewER (being an exact method) always has an error rate of 0%.

Progressive-Query-Baseline

A common approach employed in state-of-the-art progressive entity resolution methods [Whang et al., 2013; Papenbrock et al., 2015a; Simonini et al., 2018] is based on the *sorted neighborhood*. The basic idea is to sort all records according to an attribute that can capture their similarity (e.g., price of products), then to slide a window from the head to the tail of the sorted list, and to progressively compare all the pairs of records that fall within that window, i.e., the *neighborhood*.

The method as originally proposed [Whang et al., 2013] starts with a window of size $w = 2$ and then, after each iteration over the entire list, increases the size of the window (to $w = 3$, $w = 4$, etc.). The main problem with this method is that it does not satisfy the *correctness* and *monotonicity* conditions of Definition 2.4.1: each time that w increases, new matches can be found for an entity, hence the final aggregate value may change. Yet, we can choose to set a value for w, performing a single iteration over the sorted list of records and avoiding this problem (the disadvantage is that we need to pre-specify w).

To measure the quality of the progressive entity emission in an entity resolution on-demand setting of the sorted neighborhood method, we sort all records by the attribute employed in the ordering clause (`megapixels` for SIGMOD20, `price` for Altosight, and `amount` for Funding). Then, we employ $w = 10$ and $w = 100$ to represent two opposite scenarios [Papadakis et al., 2016]: the former setting favors efficiency over recall, while the latter does the opposite.

Regarding the queries, we consider only the basic query with the GROUP BY ENTITY and ORDER BY predicates: queries with selection predicates could be simply applied to the progressively generated entities, but they would not affect the entity emission order, which is what we want to assess here. As resolution functions for the ordering attribute, we consider both AVG and VOTE.

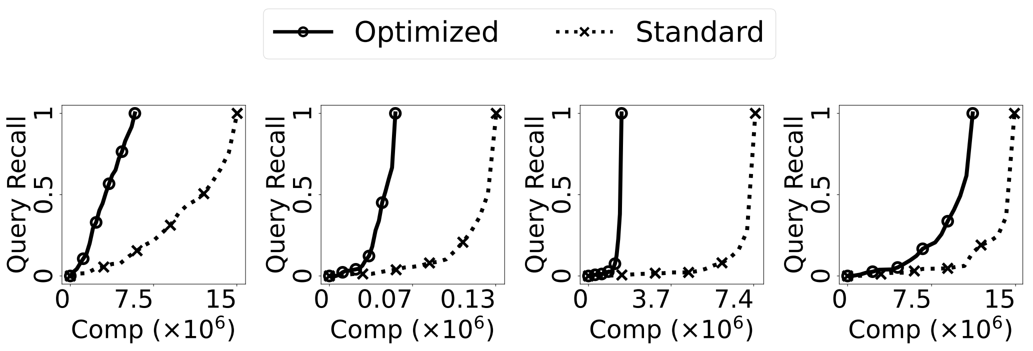
As in the previous experiment, we report the error rate ($Err@k$) of Progressive-Query-Baseline measured on the *first k* emitted entities. We mark an entity as erroneous only if the value of the ordering attribute is different from the ground truth; thus, if errors affect other attributes, we do not consider them. We did not notice any significant difference between ascending and descending ordering and report only the former. The results are shown in Figure 2.13; note that BrewER is an exact method, hence its error rate is always 0%.

In SIGMOD20, the intra-cluster variance for the ordering attribute (`megapixels`) is very low, hence the error rate with AVG (VOTE) is under 4% (1%) for the first 100 emitted entities, rising up to 8% (2.5%) for the first 1000 emitted entities. On Altosight, Progressive-Query-Baseline always fails to emit the first entity correctly ($Err@1 = 100\%$); with VOTE on the first 100 (1000) entities, the error rate is at least 60% (75%); with AVG, near to 100% for $k \in (80, 1000)$. On Funding, Progressive-Query-Baseline is correct only for $k \leq 10$ with VOTE, with high error rates (at least 50%) for all the other settings.

All these errors occur because Progressive-Query-Baseline (as all the progressive entity resolution methods) does not keep track of the value of a resolved entity while resolving it (e.g., as BrewER does through the priority queue). Hence, Progressive-Query-Baseline is not a reliable technique for entity resolution on-demand.

2.7.3 Discordant Ordering Queries

Algorithm 2.1 presents an optimized version to manage the special yet frequent case of discordant ordering (described at the end of Section 2.5.2). To evaluate its performance, we employ the same settings of Section 2.7.1, with



(a) SIGMOD20 (b) SIGMOD21 (c) Altosight (d) Funding

Figure 2.14: Progressive recall with discordant ordering queries.

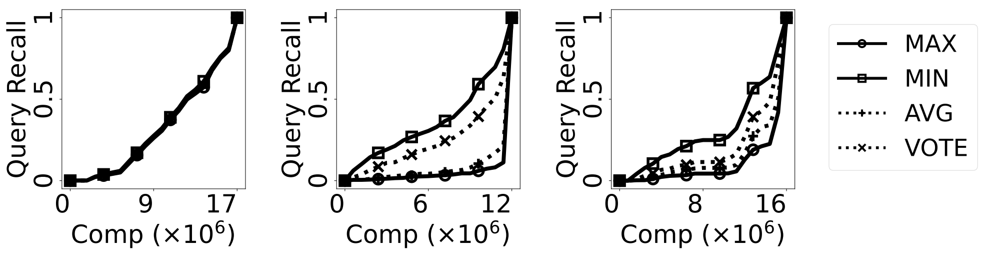
one major difference: the randomly generated queries have MAX/ASC or MIN/DESC as combinations of the aggregate function for the ordering value (i.e., MAX or MIN) and ordering mode (i.e., ASC or DESC).

Figure 2.14 compares the progressive recall of both the standard Algorithm 2.1 and the optimized version for discordant ordering. On all datasets, the optimized version terminates the queries by saving a significant amount of comparisons, up to four times compared to the standard Algorithm 2.1 on Altosight (Figure 2.14c).

We observe that for SIGMOD21 (Figure 2.14b), Altosight (Figure 2.14c), and Funding (Figure 2.14d), the recall curve is much more flat at the beginning of the plot and much steeper at the end, compared to the non-discordant-ordering case of the previous experiment (Figure 2.12). This is because with MAX/ASC and MIN/DESC queries (i.e., discordant ordering queries), when considering the head element of the priority queue, if a match is found it determines the re-insertion of the element in a lower position in the queue. This entails a higher average delay in the emission of the entities for SIGMOD21, Altosight, and Funding (Figures 2.14b, 2.14c, and 2.14d, respectively).

Differently, SIGMOD20 is only marginally affected by that phenomenon. This can be explained by the fact that in SIGMOD20 the variance of the values for the `megapixels` attribute is very low within each cluster of entity records (i.e., most of the entities have records with similar ordering values). On the other hand, the price values within a single entity in Altosight may have a high variance (e.g., due to special offers).

Finally, no significant differences can be found between conjunctive (in Figure 2.14) and disjunctive queries, as when employing the standard Algorithm 2.1.



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 2.15: Progressive recall varying aggregate functions.

2.7.4 Experiments with Aggregate Functions

In Section 2.7.1 and Section 2.7.3, only MIN and MAX have been considered. The goal of this experiment is to evaluate BrewER with a set of different aggregate functions. We set ASC as ordering mode and we run each query of the batch with the following aggregate functions: MAX, MIN, AVG, and VOTE. The batch of 20 AND queries is generated as explained in Section 2.7.1. In this experiment, MAX represents the discordant case and the optimized version is not employed.

The plots are in line with the previously observed results in Figures 2.12 and 2.14. SIGMOD20 does not present relevant variations: the performance by changing aggregate functions is almost unaltered (Figure 2.15a). Again, this can be explained by the little variance that the ordering attribute assumes among records belonging to the same entity. On the other hand, when the variance is high a significant difference in the behavior of the aggregate function is observed, as shown for the other datasets in Figures 2.15b and 2.15c.

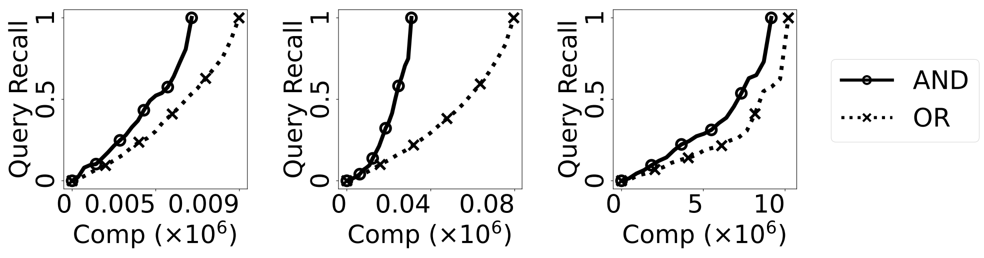
2.7.5 Performance with Blocking

With this experiment, we want to evaluate if and how the performance of BrewER changes by employing blocking. Due to its small size, SIGMOD21 is not considered for this experiment. We employ JedAI [Papadakis et al., 2019; Papadakis et al., 2020], which is based on a completely unsupervised blocking approach. We use its standard configuration based on *token blocking* and *meta-blocking* [Papadakis et al., 2020].

Table 2.4 reports recall, precision, and F_1-score of the produced candidate pairs. The goal of blocking is to reduce superfluous comparisons, while preserving as many true positives as possible; hence, it is typical to have high recall and low precision in this phase [Christen, 2012a]. The final recall

Table 2.4: Characteristics of the unsupervised blocking.

Dataset	**Recall**	**Precision**	F_1
SIGMOD20	0.933	0.407	0.567
Altosight	0.999	0.056	0.107
Funding	0.966	0.014	0.028



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 2.16: Progressive recall with unsupervised blocking.

and precision of the entity resolution process is determined ultimately by the quality of the matching function adopted, which is not evaluated here.

The queries have been synthesized as explained in Section 2.7.1 and the results are presented in Figure 2.16. By employing blocking we see a huge reduction of required comparisons for all datasets compared to the all-pairs solutions of Figure 2.12 (up to 200 times for Altosight). As for the progressive recall, with SIGMOD20 and Altosight the curve for the AND queries is much steeper than the one for the OR queries. This happens because conjunctive queries allow to filter out blocks appearing in connected components whose records do not satisfy every predicate of the query, as explained in Section 2.5.1.

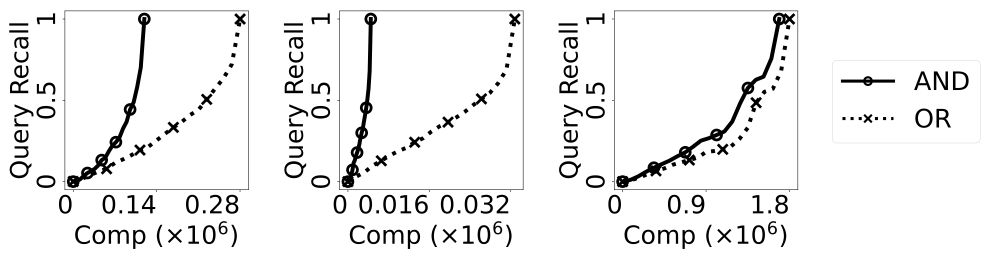
Differently, with Funding, the difference between conjunctive and disjunctive queries is less evident. This is due to to the high intra-block variance of the selection attribute values, which limits the efficacy of the preliminary block filtering.

In addition to unsupervised blocking, we also want to evaluate manually-devised blocking strategies (for simplicity, *manual blocking*). We report in Table 2.5 the characteristics of the best configuration we could find for each dataset. Yet, for Funding, the blocking strategy yields a low recall. However, it represents a plausible real-world scenario, which gives the opportunity to study whether BrewER is affected by an aggressive blocking strategy.

As depicted in Figure 2.17, no significant differences in performance can be detected, apart from the number of comparisons depending on the preci-

Table 2.5: Characteristics of the manual blocking.

Dataset	Recall	Precision	F_1
SIGMOD20	0.993	0.014	0.028
Altosight	0.999	0.086	0.158
Funding	0.586	0.023	0.044



(a) SIGMOD20 (b) Altosight (c) Funding

Figure 2.17: Progressive recall with manual blocking.

sion of the blocking function. Details about the adopted blocking strategies are illustrated below.

For SIGMOD20, for each record two keys are extracted from `brand` and `model` attribute values as follows. The first key is extracted by removing punctuation marks and numeric characters from the two attributes and concatenating the remaining strings. The second key is generated with the numerical characters of `model`. Then, these two keys are employed to build an inverted index of the records: each key corresponds to a block and two records are indexed together in the same block if they share at least one key.

For Altosight, for each record a key is generated by concatenating `brand` and `size` values (if both attribute values are different from null) and removing white spaces. The key is then used for creating the blocks as for SIGMOD20. The same criterion could be applied to its subset SIGMOD21.

For Funding, two keys are considered for each record. Firstly, if the `name` value is not null, up to two tokens before the first comma are extracted (this is done to remove organization suffixes); then, punctuation marks, numeric characters and white spaces are removed to yield the final key. A second key is extracted from the `address` value (if not null) by removing punctuation marks, numeric characters and white spaces. As for the other datasets, these keys are employed for creating the blocks.

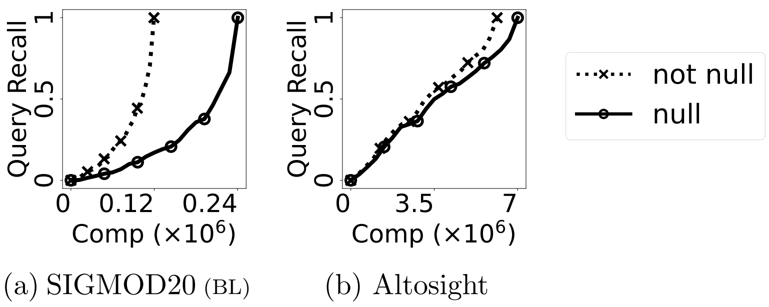


Figure 2.18: Progressive recall with missing values.

2.7.6 Performance with Missing Values

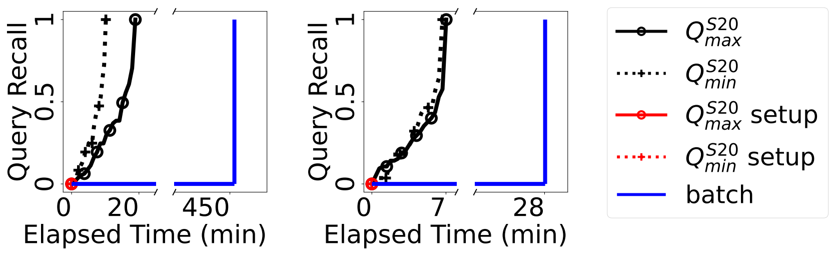
With this experiment, we want to evaluate the impact of null values in the ordering attribute. We consider only SIGMOD20 (with blocking) and Altosight, since SIGMOD21 and Funding do not present null values in their ordering attribute.

Thus, 16.2k and 1.5k additional records with null values have been added to SIGMOD20 and Altosight, respectively, for this experiment. In SIGMOD20, the new records are scattered among around 6k entities, while in Altosight only on 30 entities.

The experiment results are reported in Figure 2.18. For Altosight (Figure 2.18b), the progressive recall of the queries executed on the dataset with null values has a curve similar in shape to the one generated without considering the null values. As a matter of fact, only 30 entities have records with null values, but none of them has a final resolved ordering value equal to null: their priorities in the priority queue do not change. The only difference is the slightly higher number of comparisons needed for the additional records with null values.

Differently, Figure 2.18a showcases the negative effect of the high number of entities with null ordering value of SIGMOD20. Besides the higher number of comparisons needed for resolving the entire dataset (which has 16.2k records more than the dataset without null values), the progressive recall curve is more flattened in presence of null values for the first circa 90% of the comparisons, and steeper at the end.

This is due to the fact that many entities have an ordering value equal to null (i.e., all ordering values of their records are null): they have the lowest priority in the priority queue, so they are compared, resolved and emitted only at the end of the processing.



(a) SIGMOD20 (BL) (b) SIGMOD21

Figure 2.19: Query execution runtime in BrewER.

2.7.7 Runtime Evaluation

We finally want to assess the runtime performance of BrewER in a real-world scenario, by employing a state-of-the-art matching function and measuring the progressive recall as a function of the actual time needed for answering a query.

We design the experiment as follows. We consider a large dataset with blocking and a small one without blocking: the former is SIGMOD20, the latter is SIGMOD21. Then, from the batch of disjunctive queries of Section 2.7.1, we select for each dataset two queries: one yielding the largest result set, the other yielding the smallest result set (see Table 2.2 for the size of the query results). Thus, a total of four queries are considered: Q^{S20}_{max} and Q^{S20}_{min} for SIGMOD20, and Q^{S21}_{max} and Q^{S21}_{min} for SIGMOD21. As a matching function we employ a pre-trained deep learning classifier built with DeepMatcher [Mudgal et al., 2018] by exploiting its *hybrid* model, which we found to achieve good performances on our datasets.

The results are shown in Figure 2.19, which reports for each query the average runtime of ten executions. The plots also report the runtime required for cleaning the entire dataset with a traditional batch entity resolution method (blue line). For all datasets, the correct results start to be emitted after the first few minutes of execution.

For instance, on SIGMOD20 (Figure 2.19a), with BrewER users receive 22 and 31 entities after only two minutes for Q^{S20}_{max} and Q^{S20}_{min}, respectively. Instead, with the complete entity resolution process with a batch method users would wait 8 hours, even if the dataset has “only” thirteen thousand records and blocking is employed. A similar behavior is observable also on SIGMOD21 (Figure 2.19b).

Finally, the overhead time required by BrewER (i.e., for generating and executing the seed query and to initialize the priority queue) is negligible compared to the overall execution time of the query. In particular: *(i)* the startup time for BrewER is circa 4 and circa 0.1 seconds for SIGMOD20 and SIGMOD21, respectively; *(ii)* the average overhead introduced by BrewER for each comparison is 0.01 milliseconds, while the average runtime for comparison of the matching function is 2.7 milliseconds.

Chapter 3

Sloth

This chapter provides an extensive description of Sloth, a novel solution designed to efficiently detect the largest overlap between two tables. Sloth has been presented and described in a dedicated research paper, to be presented at SIGMOD 2024 [Zecchini et al., 2024]. The code for Sloth is openly available on GitHub[1].

First, Section 3.1 provides an overview of the dataset discovery task, which plays a fundamental role for data integration in the ELT scenario, quickly illustrating some relevant research directions in this field and the related state-of-the-art solutions. Then, Section 3.2 describes the phenomenon of overlapping tables, introducing the idea of the *largest overlap* between two tables adopted by Sloth. After discussing why existing techniques cannot provide a solution for determining the largest overlap (Section 3.3), the problem is formalized in Section 3.4. Section 3.5 describes in detail the algorithms for the detection of the largest overlap between two tables implemented by Sloth and the overall functioning of the system, while Section 3.6 reports the results and the outcomes of its experimental evaluation.

3.1 Dataset Discovery

Dataset discovery [Paton et al., 2023] is the task of detecting datasets in a corpus related to a dataset at hand, according to specified relatedness criteria. Dataset discovery has become extremely important as a preliminary step to data integration with the rise of very large table corpora, such as tables on the Web or in data lakes. Indeed, the Web contains a huge amount of structured data in tabular form. In 2008, Cafarella et al. were already able to retrieve more than 14 billion HTML tables from the Web, of which more than 150

[1] `https://github.com/dbmodena/sloth`

million contained high-quality relational data [Cafarella et al., 2008]. For instance, the English version of Wikipedia alone contained over 2 million tables as of September 2019 [Bleifuß et al., 2021b], and more than 10 million tables have been made available on GitHub over years [Hulsebos et al., 2023]. A similar situation occurs in data lakes, which can contain many tables and pose serious scalability concerns [Srinivas et al., 2023]. In these scenarios, data scientists and practitioners usually have a table and require efficient solutions to retrieve in an automated way tables from the corpus related to it, so that they can enrich its information value, e.g., inserting in the table additional entities or further attributes about the existing ones.

The first formal definition in literature of *related tables* for dataset discovery was provided by Das Sarma et al. in 2012. According to this definition, two tables are considered to be related if they can be seen as the result of applying two different queries on the same virtual table [Das Sarma et al., 2012]. Such queries can be selections, projections, or combinations of the two. The authors also provide definitions for two particularly relevant types of related tables, i.e., *unionable tables* and *joinable tables*, relying on the concepts of *entity complement* and *schema complement*, respectively. Two unionable tables are therefore seen as the result of applying two selection predicates on the virtual table, hence producing two complementary sets of entities with a similar schema. Similarly, two joinable tables are considered to be the result of applying two projection predicates on the virtual table, obtaining therefore two complementary sets of attributes describing the same entities.

The discovery of these two types of related tables in large Web table corpora or in data lakes represents the main research direction in this field, with a plethora of efficient solutions proposed for detecting unionable tables [Cafarella et al., 2009; Lehmberg and Bizer, 2017; Nargesian et al., 2018], joinable tables, such as LSH Ensemble [E. Zhu et al., 2016], Aurum [Castro Fernandez et al., 2018], JOSIE [E. Zhu et al., 2019], Pexeso [Y. Dong et al., 2021], MATE [Esmailoghli et al., 2022], and DeepJoin [Y. Dong et al., 2023], or both, such as D^{3}L [Bogatu et al., 2020] and Juneau [Zhang and Ives, 2020], also providing insights about their subsequent integration, as done by ALITE [Khatiwada et al., 2022].

For instance, JOSIE [E. Zhu et al., 2019] is one of the most influential solutions for the detection of the top-k table columns in a large table corpus that can be joined with a given query table T_Q on a specified column Q. In particular, JOSIE represents both Q and all columns from the tables in the corpus as sets of cell values, then retrieves the k columns with the largest *overlap set similarity* [D. Deng et al., 2018] to Q. The overlap set similarity is simply computed as the size of the intersection of the two sets (i.e., the result is composed of the k columns with the most distinct cell values in common with Q). As previously done by LSH Ensemble [E. Zhu et al., 2016], this measure is adopted since it is not biased against sets of different sizes, differently from the widely used Jaccard similarity [Jaccard, 1912], which is instead computed as the size of the intersection over the size of the union of the two sets. Nevertheless, while LSH Ensemble produces an approximate solution, JOSIE relies on the exact computation of the overlap set similarity to measure joinability. JOSIE operates using an *inverted index*, which for each distinct cell value builds a *posting list* composed of pointers to the sets (i.e., columns) in which it appears. To efficiently retrieve the top-k joinable columns, JOSIE estimates the set intersection size and uses this estimation to prioritize the reading of sets or posting lists, favoring at every step the operation that produces the greatest benefit (hence minimizing the number of computed set intersections).

MATE [Esmailoghli et al., 2022] represents instead the state-of-the-art solution for detecting multi-column joins, retrieving the top-k tables with the greatest joinability (i.e., equi-join cardinality) given a query table T_Q and a set of columns Q out of it. MATE uses a novel hash function, named Xash, to produce a fixed-sized hash value called *super key* for each table row in the corpus. Since the combination of $|Q|$ columns that maximizes the joinability of a table with T_Q on Q is not known a priori (in particular, it is needed to detect the one-to-one attribute mappings), the super key masks all possible combinations of cell values present in a row, acting similarly to a Bloom filter [Bloom, 1970] to quickly discard useless candidates. Each cell value in the row is encoded into a distinguishable fixed-sized hash value by encoding its syntactic features using a minimal number of bits set to one, to avoid collisions as much as possible. In particular, for each cell value Xash encodes its least frequent characters, together with their location inside the cell value itself, and its length, performing a final bit rotation step to further reduce the probability of false positives. The obtained hash values are then aggregated into the super key through bit-wise OR, making it possible to quickly check if a row may contain the combination of cell values present in a row of the join columns Q by a simple comparison to its super key.

A related research direction which is gaining more and more popularity and achieving notable results in the latest years is *representation learning* for tabular data. In particular, a wide range of solutions [Badaro et al., 2023] is based on the transformer architecture [Vaswani et al., 2017], such as TURL [X. Deng et al., 2020], TaPas [Herzig et al., 2020], TaBERT [Yin et al., 2020], and TABBIE [Iida et al., 2021]. These solutions rely on the pre-training/fine-tuning approach and, differently from other table representation learning techniques such as Table2Vec [L. Deng et al., 2019], learn deep contextualized representations for different table components (e.g., caption, headers, cell entities, etc.). One of the main challenges to overcome is to capture the structured format of the table, for which different techniques have been designed. For instance, TURL [X. Deng et al., 2020] does this through a *visibility matrix* that acts as an attention mask, ensuring that each component can only aggregate information from other structurally related components of the table during the self-attention calculation. Thus, while the caption tokens are visible to all components, only cell entities in the same row or the same columns are visible to each other.

Finally, it is worth mentioning the key role played in dataset discovery research by several widely adopted large table corpora generated over years, such as WikiTables [Bhagavatula et al., 2015], the Dresden Web Table Corpus [Eberius et al., 2015], WDC Web Table Corpora [Lehmberg et al., 2016], GitTables [Hulsebos et al., 2023], or the more recent WikiDBs [Vogel and Binnig, 2023].

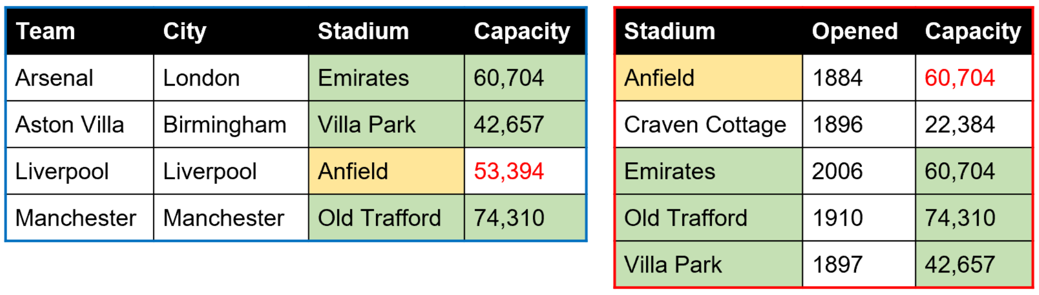
Despite dataset discovery research has produced many notable results and is gaining more and more attention in the latest years, some relevant issues still need to be explored. Indeed, while a significant effort was put into the design of efficient techniques for the discovery of unionable and joinable tables, other possible definitions of relatedness have barely been touched by the existing literature. This is for instance the case of *duplicate tables*. We decided therefore to look at this problem from a novel perspective, delving deeper into the phenomenon of overlapping tables.

3.2 Overlapping Tables

Both on the Web and in data lakes, it is possible to detect much redundant data in the form of largely overlapping pairs of tables. In particular, we focus on the *largest overlap* between the two tables, i.e., their largest common *rectangular* subtable, as depicted in Figure 3.1. Detecting the largest overlap is not trivial: the nature of tabular data allows changing the order of columns and rows (Figure 3.1b), making this task computationally challenging.

In many cases, the largest overlap between two tables is not accidental, but gives significant insights about the relatedness of the tables and the quality of the conveyed information. Let us consider a real example from Wikipedia. The tables presented in Figure 3.2, reporting the players of a US college football team selected in the 1955 NFL draft, appear in a page describing the previous season of the team[2] and in a page collecting the infor-

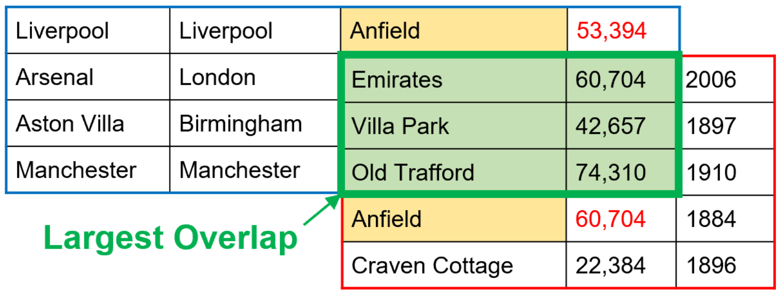
[2]`https://en.wikipedia.org/?oldid=1153086262`



Team	City	Stadium	Capacity
Arsenal	London	Emirates	60,704
Aston Villa	Birmingham	Villa Park	42,657
Liverpool	Liverpool	Anfield	53,394
Manchester	Manchester	Old Trafford	74,310

Stadium	Opened	Capacity
Anfield	1884	60,704
Craven Cottage	1896	22,384
Emirates	2006	60,704
Old Trafford	1910	74,310
Villa Park	1897	42,657

(a) A pair of tables about football teams and stadiums.



Liverpool	Liverpool	Anfield	53,394	
Arsenal	London	Emirates	60,704	2006
Aston Villa	Birmingham	Villa Park	42,657	1897
Manchester	Manchester	Old Trafford	74,310	1910
		Anfield	60,704	1884
		Craven Cottage	22,384	1896

(b) The largest overlap between the two tables.

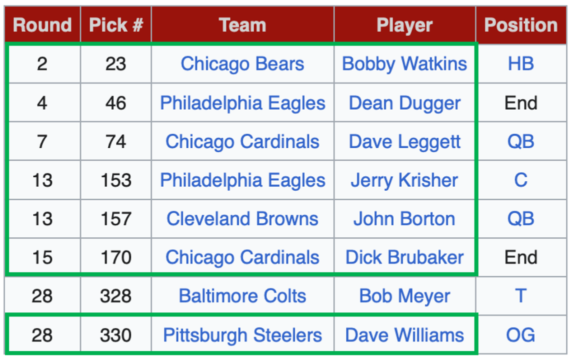
Figure 3.1: Example of largest overlap between tables.

mation about all draft picks during the team history[3], respectively. As can be noticed from the headers, the order of the columns is different and the two tables, which in principle should convey the same information, present some inconsistencies. In particular, different conventions for the player positions and a conflict on the surname of a player (*Meyer* vs. *Myers*) cause the removal of an entire column and an entire row from the largest overlap (since in this case forcing the inclusion of their common elements would lead to a smaller overlap size).

As made evident by this real-world example, the ability to detect the overlap between two tables, and in particular to retrieve pairs of highly similar tables, defined as *matching* or *duplicate* tables, can lead to several benefits. First, it allows checking the consistency of the information conveyed by the tables, pointing out cases of incompleteness or inconsistencies to be fixed by the editors. This aspect has a paramount importance in the context of an encyclopedia, such as Wikipedia, where the information contained in the tables should always be correct, complete, and updated.

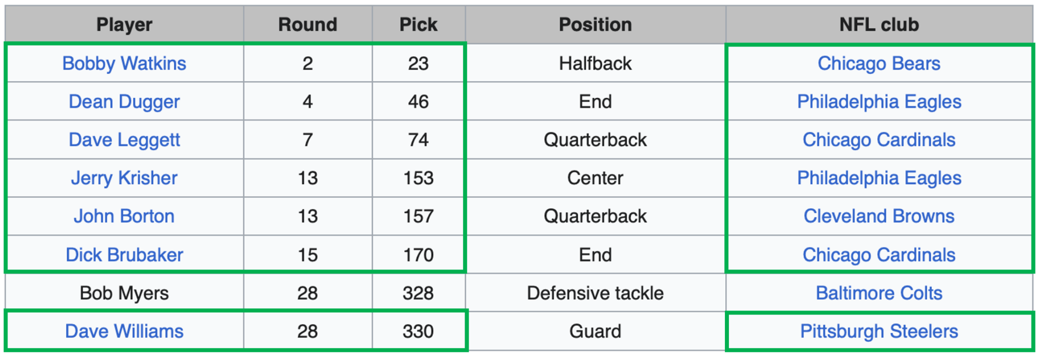
In fact, tables on Wikipedia have a very dynamic existence [Bleifuß et al., 2021b]: they are frequently edited or updated, moved within their page or

[3]`https://en.wikipedia.org/?oldid=1160019836`



Round	Pick #	Team	Player	Position
2	23	Chicago Bears	Bobby Watkins	HB
4	46	Philadelphia Eagles	Dean Dugger	End
7	74	Chicago Cardinals	Dave Leggett	QB
13	153	Philadelphia Eagles	Jerry Krisher	C
13	157	Cleveland Browns	John Borton	QB
15	170	Chicago Cardinals	Dick Brubaker	End
28	328	Baltimore Colts	Bob Meyer	T
28	330	Pittsburgh Steelers	Dave Williams	OG

(a) 1954 Ohio State Buckeyes football team/1955 NFL draftees.



Player	Round	Pick	Position	NFL club
Bobby Watkins	2	23	Halfback	Chicago Bears
Dean Dugger	4	46	End	Philadelphia Eagles
Dave Leggett	7	74	Quarterback	Chicago Cardinals
Jerry Krisher	13	153	Center	Philadelphia Eagles
John Borton	13	157	Quarterback	Cleveland Browns
Dick Brubaker	15	170	End	Chicago Cardinals
Bob Myers	28	328	Defensive tackle	Baltimore Colts
Dave Williams	28	330	Guard	Pittsburgh Steelers

(b) List of Ohio State Buckeyes in the NFL draft/1955 NFL draft selections.

Figure 3.2: Example of coexisting matching tables in related Wikipedia pages, presenting a different column order, a different convention for the Position column, and an inconsistency on the name of a player (*Bob Meyer* vs. *Bob Myers*). The cells included in the largest overlap are highlighted by green squares.

to another page, copied to related pages (e.g., when a topic is analyzed at different levels of detail, a table often appears both in the most generic page and in more specific ones) or somewhere else (even just to use them as a template), with frequent episodes of carelessness, conflicts among editors [Bykau et al., 2015], or vandalism [Potthast et al., 2008]. Considering this dynamism and the heterogeneity of the community of Wikipedia editors (which is composed of a huge number of people from all over the world), despite various attempts to automate the detection and the resolution of some kinds of inconsistencies [Sottovia et al., 2019; Barth et al., 2023], it can be very difficult to assess the quality of the tabular data appearing there, especially on less popular pages.

The tabular form is widely used in Wikipedia to represent data. In fact, limited to its English version, more than 60 million tables have appeared there throughout its history up to September 1, 2019 [Bleifuß et al., 2021b]. Among the 2.13 million tables existing at the time of the latest snapshot, we surprisingly discover that about 6.5 million table pairs present an overlap equal to at least half of the area of the smaller table, for an estimated redundancy of 63.49 MB. Even more surprisingly, we found that Wikipedia contains a huge amount of 5.9 million pairs of coexisting tables with identical content, highlighting the diffusion of copy-and-paste practices in such a scenario and the potential of centrally serving data for Wikipedia[4].

More generally, a scenario where a table can be duplicated at a certain point in time, with an independent development for the different copies, represents one of the main factors for the generation of inconsistencies. This is a common scenario that often occurs with enterprise data as well. For instance, when data scientists retrieve datasets from the enterprise's data lake, perform transformations (e.g., join, wrangling, etc.) for their analysis, then store back the new datasets into the data lake.

Depending on the context, a user might desire to leverage on the detected largest overlap to ensure the consistency of the information present in duplicate tables through operations of *data cleaning* [Ilyas and Chu, 2019] and *change propagation* [Bleifuß et al., 2018], or it might be more convenient to directly prevent the rise of inconsistencies by eliminating this redundancy. In fact, avoiding redundancy not only allows to save disk space in the case of data lakes, where this phenomenon is quite frequent, but also to lighten the workload for website editors, who would just have to focus on a single consistent table instead of performing every editing multiple times, exposing to the risks shown by the example above. Whatever the purpose among those mentioned, one must first detect such duplicate tables.

Therefore, we designed Sloth, a novel method to determine the largest overlap between a given pair of tables, i.e., the maximal contiguous rectangular area of identical cells that can be achieved by reordering columns and rows of both tables. Sloth introduces the first algorithm designed for this task. First, our algorithm detects the pairs of attributes across the two tables that share some cell values. By combining these pairs of attributes, it is possible to obtain a complete overview of all potentially non-empty overlaps existing between the tables (i.e., the candidates to be the largest one) in the form of a *lattice* [Birkhoff, 1940]. The combined pairs determine an upper bound for the area of the candidates. Thus, our algorithm aims to exploit this bounding mechanism to prioritize candidates and detect the largest overlap as soon as possible, minimizing the number of candidates for which we

[4]All details about this analysis are presented in Section 3.6.3.

need to compute the actual area.

This task is computationally challenging, so we also propose a greedy variant for the algorithm based on *beam search* [Lowerre, 1976] to deal with critical pairs, when the exact algorithm struggles to produce a result in a reasonable time for the user.

Our experimental evaluation on real-world datasets assesses the efficiency of Sloth and the quality of the results produced by the greedy algorithm. Moreover, it highlights some relevant real-world use cases for Sloth, such as the detection of highly overlapping tables in a very popular context like Wikipedia, the recognition of potential copying between tables from different sources [X. Li et al., 2012], and the discovery of candidate multi-column joins in a corpus of relational tables.

3.3 Comparison with Related Work

This section discusses why existing techniques cannot provide a solution to the largest overlap detection problem, hence the necessity of Sloth. This analysis moves in four main directions: *(i)* the previously described algorithms for the efficient discovery of joinable, unionable, and related tables; *(ii)* the existing solutions facing the problem of detecting duplicate tables; *(iii)* the case of partial n-ary inclusion dependencies from data profiling research [Abedjan et al., 2018]; *(iv)* the use of similarity measures based on the overlap between two matrices in different scientific domains.

Dataset Discovery

As stated in Section 3.1, a plethora of algorithms has been designed for the efficient discovery of related tables, with a special attention to the cases for unionability and joinability. These solutions cannot compute the actual value of the largest overlap, but in some cases they can produce an upper bound for it. Thus, some of them might be exploited to scale to large table corpora, passing to Sloth only the most promising pairs to evaluate.

For example, JOSIE computes joinability on a single column using the set semantics, hence it is impossible to use it for obtaining the actual size of the largest overlap. Nevertheless, if we consider the entire tables under the bag semantics instead of the single columns, the similarity would reflect the number of common cells between the two tables, which represents an upper bound for the size of the largest overlap. Thus, such an adapted version of JOSIE might be used to detect for a query table the top-k tables that are most promising to present the maximum largest overlap, then pass them to SLOTH to compute the actual values.

MATE requires the user to provide a set of columns as input; then, it retrieves all possible tables with a set of columns that might join with the user-provided set. Thus, MATE cannot be easily employed to detect the largest overlap, since the set of columns that yields the largest overlap is not known by the users up-front.

Duplicate Table Detection

While the existing literature put much effort on the efficient detection of joinable or unionable tables, the task of detecting duplicate tables has been mostly overlooked by the existing research, where it was presented only in some specific or limited scenarios.

In particular, Koch et al. provide the first definition to this problem and propose to use XASH to detect duplicate tables in data lakes, designing a pipeline for both query table and data lake deduplication scenarios [Koch et al., 2023]. The authors define two tables as duplicates if they contain the same sets of tuples, possibly after permuting the columns of one table. Thus, they only tackle the basic cases of perfect duplicates or row containment, ignoring both column containment and mostly those cases where the overlap misses cells from both tables, which pose the most significant challenges and usually provide the most meaningful insights.

A similar task is performed by Bleifuß et al. to detect matching tables across subsequent versions of a Wikipedia page [Bleifuß et al., 2021a]. However, the proposed approach, relying on a multi-stage matching based on Jaccard similarity (also in a relaxed form), is specifically designed for one-to-one matches among a limited number of tables sharing the same context, also exploiting specific aspects such as the position of the tables inside the page, hence not generalizable.

It is worth noting that in the literature the expression *table matching* has also been employed to refer to the task of matching Web tables with knowledge bases [Limaye et al., 2010; J. Fan et al., 2014; Ritze et al., 2015], i.e., to annotate their cells, rows, and columns with entities and properties from a knowledge base, such as YAGO [Pellissier Tanon et al., 2020] or DBpedia [Bizer et al., 2009]. By identifying a common link through the knowledge base, these methods could be adapted to detect overlaps among tables. The major drawback of this approach would be that all the cells that are not linked with an entity in the knowledge base (e.g., YAGO) would not be considered for computing the table overlaps, hence making it highly inaccurate in many scenarios.

Partial *n*-ary Inclusion Dependencies

Inclusion dependencies (INDs) are a kind of data dependency expressing that the set of tuples of one column combination is contained in the set of tuples of another column combination, hence allowing to discover foreign keys among relational tables. INDs are a prominent topic in data profiling research and many approaches have been proposed to detect them efficiently [De Marchi et al., 2002; Tschirschnitz et al., 2017; Kaminsky et al., 2023]. In particular, our definition of largest overlap shows affinity with the case for *partial* n*-ary INDs*. Partial INDs [Lopes et al., 2002] allow a defined amount of tuples to violate the containment constraint (since the largest overlap does not need to cover all rows), while *n*-ary INDs [De Marchi and Petit, 2003; De Marchi et al., 2009; Papenbrock et al., 2015b] consider combinations covering more than one column (as usually happens for the largest overlap). Therefore, an algorithm for detecting partial *n*-ary INDs should return the largest overlap among the detected dependencies. Unfortunately, even if some algorithms for detecting INDs have been extended to separately deal with both partial and *n*-ary INDs [Dürsch et al., 2019], no solution has been proposed yet to find INDs presenting both of these features. Note that the algorithms for discovering INDs adopt the set semantics, while our problem would require the bag semantics to correctly evaluate the overlap area.

Overlap-Based Similarity Measures

While Sloth is the first solution to detect the largest overlap between tables, some related approaches in different research areas use the *largest common submatrix* to estimate the similarity between two matrices, e.g., as a distance measure between images [Amelio and Pizzuti, 2013] or for determining co-evolving proteins [Tillier and Charlebois, 2009]. Despite their affinity, the case of tables remains unique due to the possibility of changing the order of columns and rows, making this problem significantly more complex.

3.4 Largest Overlap Definition

This section aims at laying the theoretical foundations for the detection of the largest overlap between two tables, presenting a formal definition of what we mean by *table overlap* (or simply *overlap* when evident from the context), describing how to compute its area, and evaluating the computational complexity of the problem.

The definition of table overlap relies on the concept of *attribute mapping*, which is defined as follows.

Definition 3.4.1 (Attribute Mapping). Given two tables $R(X)$ and $S(Y)$, where X and Y denote their schemas (i.e., their attribute sets), an *attribute mapping* (or simply *mapping*) between $R(X)$ and $S(Y)$ is defined as a bijective function M that maps a subset of X to a subset of Y:

$$M : X_M \subseteq X \to Y_M \subseteq Y$$

Due to the bijectivity of the mapping, the attribute sets have the same size ($|X_M| = |Y_M|$), no attribute in X is mapped to more than one attribute in Y, and no attribute in Y is mapped from more than one attribute in X. Each mapping determines a table overlap. Using $\Cap$ to denote the intersection under the bag semantics (which allows duplicates), we can define the table overlap and its area as follows.

Definition 3.4.2 (Table Overlap). Given a mapping M, defined between tables $R(X)$ and $S(Y)$ considering the attribute subsets $X_M \subseteq X$ and $Y_M \subseteq Y$, the *table overlap* O_M is the bag intersection between the bags of the tuples obtained through the projection of $R(X)$ on X_M and $S(Y)$ on Y_M:

$$O_M = R[X_M] \Cap S[Y_M]$$

Definition 3.4.3 (Overlap Area). We define the *overlap area* A_M as the number of cells contained in the overlap O_M:

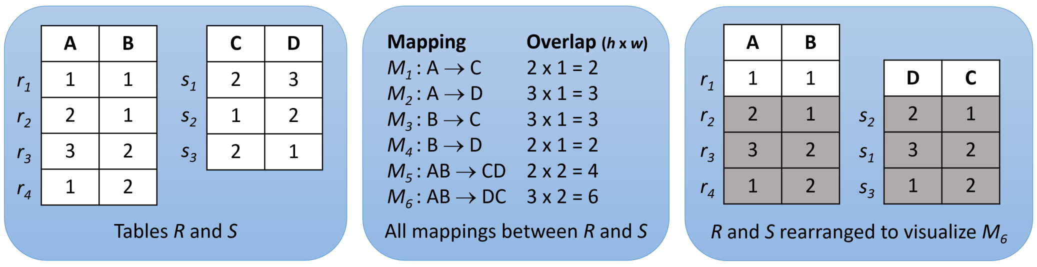
$$A_M = |X_M| \cdot |O_M|$$

where $|X_M|$ and $|O_M|$ represent the *width* and the *height* of the rectangle of overlapping cells between the two tables, respectively. We also refer to A_M as the area of mapping M.

Definition 3.4.4 (Largest Overlap). Let $\mathcal{O}$ be the set of overlaps determined by all possible mappings between $R(X)$ and $S(Y)$. We define the set of the *largest overlaps* $\mathcal{O}^* \subseteq \mathcal{O}$ as those overlaps that have the maximum area:

$$\mathcal{O}^* = \{O_{M^*} \in \mathcal{O} \mid A_{M^*} \geq A_M, \forall O_M \in \mathcal{O}\}$$

We refer to the mappings of $\mathcal{O}^*$ as *top mappings*, denoted as M^*. Note that the number of top mappings (in most cases just one) is equal to the number of largest overlaps. Figure 3.3 shows an example of two tables and the effects of different mappings.



	A	B
r_1	1	1
r_2	2	1
r_3	3	2
r_4	1	2

	C	D
s_1	2	3
s_2	1	2
s_3	2	1

Tables *R* and *S*

Mapping	Overlap (*h* x *w*)
M_1 : A → C	2 x 1 = 2
M_2 : A → D	3 x 1 = 3
M_3 : B → C	3 x 1 = 3
M_4 : B → D	2 x 1 = 2
M_5 : AB → CD	2 x 2 = 4
M_6 : AB → DC	3 x 2 = 6

All mappings between *R* and *S*

	A	B
r_1	1	1
r_2	2	1
r_3	3	2
r_4	1	2

	D	C
s_2	2	1
s_1	3	2
s_3	1	2

R and *S* rearranged to visualize M_6

Figure 3.3: Example to demonstrate the overlap between two tables determined by different mappings.

Computational Complexity

Let us consider the OVERLAP decision problem, whose instance is composed of two tables R and S and a positive integer K. OVERLAP raises an affirmative answer if there exists a subtable O common to both R and S with area $A_O \geq K$. Note that the problem considers all overlaps between R and S, not only the largest ones. It is easy to see that OVERLAP $\in$ NP, since a nondeterministic algorithm only needs to guess a subset of row and column pairs from R and S and check in polynomial time that the two obtained subtables are equal and their area $A_O \geq K$.

The classical NP-complete CLIQUE decision problem [Garey and Johnson, 1979], raising an affirmative answer if a graph $G = (V, E)$ contains a clique of at least C vertices, where $C \leq |V|$ is a positive integer, can be transformed into OVERLAP. In fact, considering the adjacency matrix $M : |V| \times |V|$ of the graph G, where $M_{i,j} = 1$ if $(V_i, V_j) \in E$ or $i = j$, G contains a clique of at least C vertices if and only if between M and $I : |V| \times |V|, I_{i,j} = 1, \forall i, \forall j$, there exists a (square) overlap O with area $A_O \geq C^2$ and the sets of indices for the columns and for the rows mapped from M are equal. The constraint on the indices discriminates between a clique and other structures determining square overlaps in M, such as bicliques. The transformation is polynomial (quadratic in the number of vertices). Hence, the OVERLAP decision problem is NP-complete.

Search Space

Every possible mapping M among the attributes of a table pair defines a table overlap. To obtain the size of the search space, we determine the number of such mappings. We can model this problem as a bipartite graph, where the attributes of each table represent one of the two nonadjacent vertex sets. For a worst-case analysis, let us assume that this bipartite graph is complete,

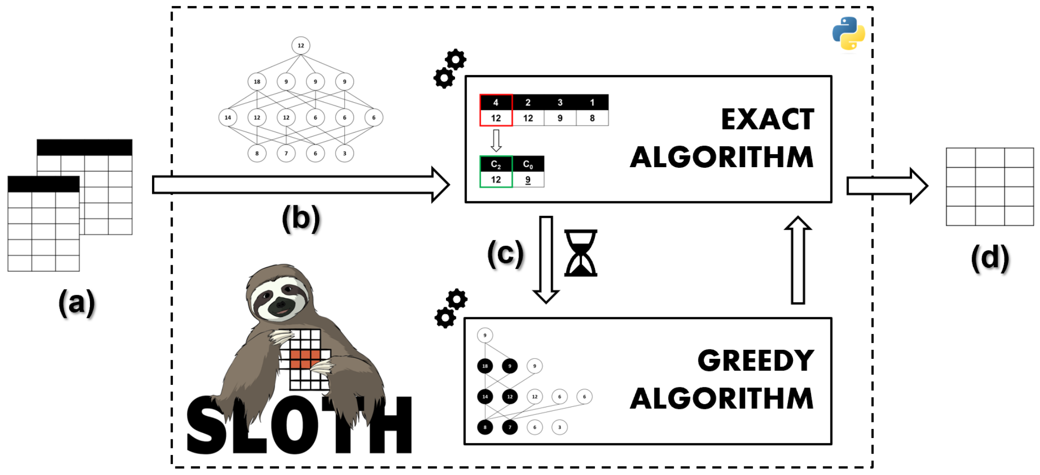


Figure 3.4: An overview of the SLOTH workflow (logo by Giacomo Pirani), from a pair of tables (a) to their largest overlap (d) through our detection algorithm (b) and its greedy variant for critical cases (c).

i.e., we can consider mapping every attribute of one table to any attribute of the other table. Each mapping now corresponds to an independent edge set, i.e., a subset of the edges so that none of these are adjacent (i.e., share a common vertex). For a complete bipartite graph with vertex sets of size $|X|$ and $|Y|$, the number of independent edge sets (a.k.a. *Hosoya index*[5]) follows the formula[6]:

$$\sum_{k=0}^{\min(|X|,|Y|)} k! * \binom{|X|}{k} * \binom{|Y|}{k}$$

Note that according to our definition a mapping does not need to cover all attributes of either table. The number corresponds to choosing two k-sized subsets out of X and Y and then permuting one side while keeping the other side fixed. This grows super-exponentially in the number of attributes of the table with fewer attributes.

3.5 Largest Overlap Detection

Figure 3.4 represents the high-level design of SLOTH. Implemented in Python, SLOTH considers as input a pair of tables (Figure 3.4a) and returns the largest overlap detected between them (Figure 3.4d). In addition to this, SLOTH can also return some additional insights on the original tables,

[5]https://mathworld.wolfram.com/HosoyaIndex.html
[6]https://oeis.org/A086885

such as the visualization for each table of the cells excluded from the result, making it easier for the user to point out the differences and the inconsistencies between the two tables.

To detect the largest overlap, SLOTH implements our novel algorithm designed for this task (described in detail in Section 3.5.1) and its greedy variant inspired by the beam search technique for the most challenging pairs of tables (illustrated in Section 3.5.2). In particular, the exact algorithm is our default choice (Figure 3.4b), with a timeout mechanism defined to spot critical cases that would not provide a result in a reasonable time, activating the greedy algorithm (Figure 3.4c). This choice is discussed in more depth in Section 3.5.2. The timeout is set by the user and is aimed at the early detection of the described critical cases. The parameter for the greedy algorithm (i.e., the beam width β, clarified in Section 3.5.2) is set to a default value selected based on our experimental evaluation (Section 3.6.2) and can be edited according to the user's needs (e.g., a faster computation or a better accuracy).

3.5.1 Exact Algorithm

This section presents our algorithm for the detection of the largest overlap(s) between two tables (Algorithm 3.1). The algorithm considers as input two tables (e.g., the two tables about football teams in Figure 3.5), denoted as $R(X)$ and $S(Y)$, and returns the set of their largest overlaps $\mathcal{O}^*$. If the two tables have no cells in common, the area of the largest overlap is equal to zero and an empty set is returned. The optional argument Δ represents the minimum area to consider the overlap as relevant and therefore to include it in the result. For instance, the user might consider the largest overlap as relevant if its area is at least equal to a certain percentage of the area of the smallest table. The definition of minimum/maximum thresholds for the width/height of the overlap is also supported, leading to the detection of the largest overlap among those complying with the defined boundaries. Since implementing this feature is straightforward, details are overlooked to avoid overloading the formalization of the algorithm. Some examples of its application to real-world use cases are provided in Sections 3.6.4 and 3.6.5.

Our algorithm needs to consider all mappings that can potentially determine the largest overlap between the two tables. These mappings are denoted as *candidates*, since they are candidates to be the top mapping. To identify the candidates, our algorithm first considers all possible *single-attribute mappings*, i.e., those mappings for which X_M is represented by a single attribute x, checking the area of the overlap between $R[x]$ and $S[M(x)]$. We call *seeds* those single-attribute mappings whose area is greater than zero, and we col-

Algorithm 3.1: Largest overlap detection algorithm

```
Input: Two tables $R(X)$ and $S(Y)$; minimum area $\Delta$ (default 0)
Output: The set of the largest overlaps $\mathcal{O}^*$
1  $\mathcal{O}^* \leftarrow \emptyset$                                   // largest overlaps
2  $Seeds \leftarrow$ findSeeds$(R, S)$
3  $\theta \leftarrow max(\Delta, Seeds[0].A)$                          // pruning threshold
4  $Levels \leftarrow$ initLevels$(Seeds)$                    // pq to generate candidates
5  $Candidates \leftarrow maxHeap(\emptyset, key = A)$       // pq to verify candidates
6  while $Candidates \neq \emptyset$ or $Levels \neq \emptyset$ do
7      while $Candidates.head().A < Levels.head().A$ do
8          $Levels, Candidates \leftarrow$ genCand$(Levels, Candidates, Seeds)$
           // generate more candidates
9      if $Candidates \neq \emptyset$ then
10         $topC \leftarrow Candidates.pop()$                      // top candidate
11         if $topC.O \neq \emptyset$ then
12             $\mathcal{O}^* \leftarrow \mathcal{O}^* \cup topC.O$             // largest overlap found!
13         else
14             $Levels, Candidates \leftarrow$
                 verCand$(R, S, topC, Levels, Candidates)$
               // verify top candidate
15 return $\mathcal{O}^*$
```

lect them in a dedicated list (Line 2), represented in Figure 3.6, where they are sorted by descending area.

Since every *multi-attribute mapping* is a combination of some single-attribute mappings, the candidates can be considered as the combinations of the seeds and modeled as the nodes of a lattice, as depicted in Figure 3.6, where every level n contains the combinations of n seeds. Moving up within the lattice increases the width of the overlap, but not necessarily its area, as its height may decrease as new columns are added. In particular, the seed with the minimum area (equal to its height) in the combination defines an upper bound for the height of the candidate, and therefore for its area. This bounding mechanism can be exploited both to prune the lattice and to prioritize the candidates based on their potential area. We define therefore the *pruning threshold* θ (Line 3), which always contains the maximum of Δ and the maximum actual area of a candidate that we know so far, which is initially the area of the first seed in the list and can be possibly updated every time we verify a new candidate, discovering its actual area.

Our algorithm leverages on the upper bound defined by the seeds to

Team	City	Stadium	Capacity
Arsenal	London	Emirates Stadium	60,704
Barcelona	Barcelona	Camp Nou	99,354
Bayern Munich	Munich	Allianz Arena	75,000
Inter Milan	Milan	San Siro	80,018
Liverpool	Liverpool	Anfield	53,394
Manchester United	Manchester	Old Trafford	74,310
Milan	Milan	San Siro	80,018
Real Madrid	Madrid	Santiago Bernabéu	81,044

Team	City	Country	Stadium	Founded
Real Madrid	Madrid	Spain	Santiago Bernabéu	1902
Milan	Milan	Italy	San Siro	1899
Bayern Munich	Munich	Germany	Allianz Arena	1900
Liverpool	Liverpool	England	Anfield	1892
Barcelona	Barcelona	Spain	Camp Nou	1899
Ajax	Amsterdam	Netherlands	Johan Cruyff Arena	1900
Inter Milan	Milan	Italy	Giuseppe Meazza	1908
Manchester United	Manchester	England	Old Trafford	1878
Chelsea	London	England	Stamford Bridge	1905
Juventus	Turin	Italy	Juventus Stadium	1897

Figure 3.5: Two input tables $R(X)$ and $S(Y)$ describing football teams.

manage two priority queues, aiming to minimize both the number of candidates that need to be materialized and those among them whose actual area needs to be computed: *(i) Levels* (Line 4), containing the representations of the levels of the lattice, used to generate the candidates incrementally; *(ii) Candidates* (Line 5), containing the generated candidates, used to progressively verify their actual area and detect the largest overlap.

In particular, we iterate on the priority queues until both of them are emptied (Line 6), terminating early as soon as all largest overlaps are detected (or only the first one, if we are only interested in their area with no need to consider their content). At each iteration, first we need to ensure that at least one of the candidates with the potential largest overlap has been generated and inserted into *Candidates* (Lines 7-8), as depicted in Figure 3.7. Next, we can check the candidate at the top of *Candidates* (Lines 9-10). If it has already been verified (i.e., its actual overlap has already been computed), none of the other candidates (among both the ones already generated and the ones yet to generate) can present a greater area, hence it is one of the

Seeds

seed	S_0	S_1	S_2	S_3	S_4
M	City ↓ City	Team ↓ Team	Stadium ↓ Stadium	Team ↓ City	City ↓ Team
A	**8**	**7**	**6**	**3**	**3**

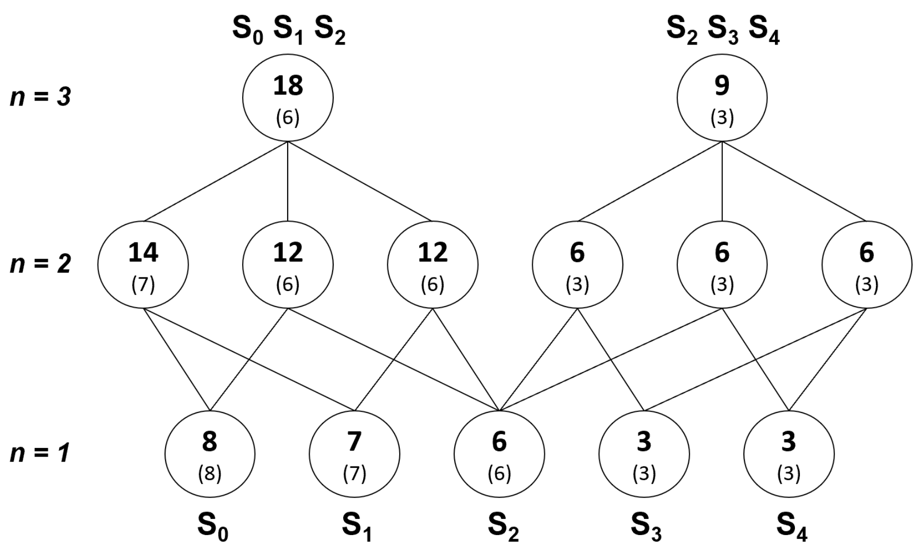


Figure 3.6: The sorted list of detected seeds and the valid candidates in the lattice generated from the seeds. For every node in the lattice we report the upper bounds for its area and its height (between brackets).

largest overlaps, and we can add it to the result set (Lines 11-12); otherwise, we need to compute its overlap (and therefore its actual area) and reinsert it into the priority queue if it can still be part of the result set (Lines 13-14).

The next sections go into the details of the different steps composing the algorithm, providing their formal representation and using the example introduced in Figures 3.5, 3.6, and 3.7 to help their understanding.

Detecting the Seeds

First, we need to detect the seeds. As stated in Line 2, this operation is delegated to the **findSeeds**() function (Function 3.2). Here the seeds, stored in a dedicated eponymous list (Line 1), are detected by verifying the area of all possible single-attribute mappings defined between the two tables (Lines 2-7). If its area is greater than zero, then a mapping is inserted into *Seeds*

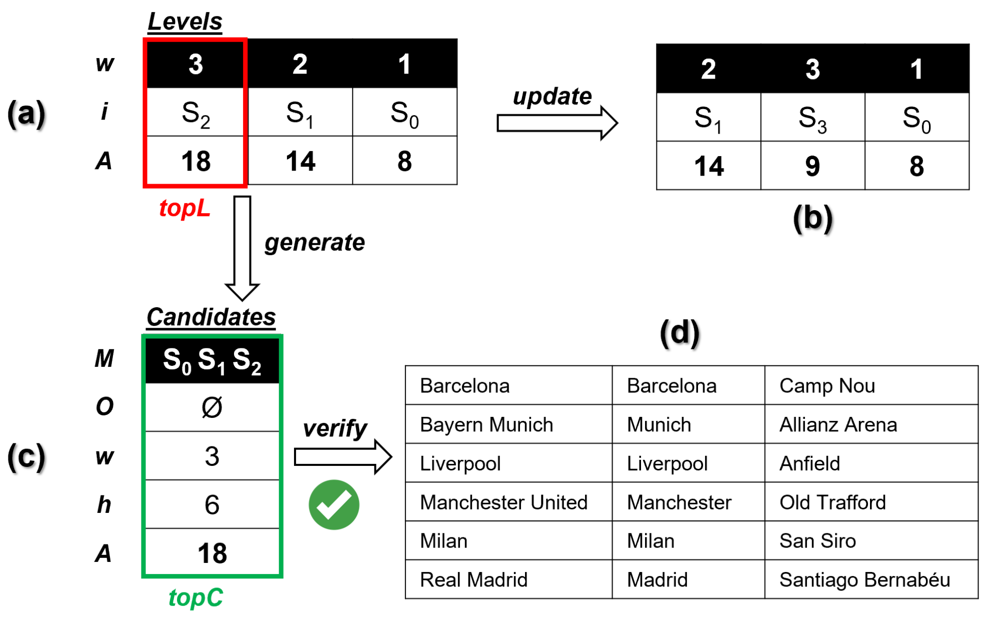
Algorithm 3.2: findSeeds() function

```
Input: The two tables $R(X)$ and $S(Y)$
Output: The sorted list of the detected seeds
1 $Seeds \leftarrow \emptyset$
2 forall $x$ in $X$ do
3     forall $y$ in $Y$ do
4         $seed.M \leftarrow M : x \rightarrow y$                // mapping
5         $seed.A \leftarrow |R[x] \Cap S[y]|$                 // area
6         if $seed.A > 0$ then
7             $Seeds.append(seed)$
8 return $Seeds.sort(A, desc)$        // sort by increasing dominance
```

(Lines 6-7), keeping track of its area. We denote the number of detected seeds as s, i.e., $s = |Seeds|$. If an attribute from one table has cells in common with multiple attributes from the other table, that attribute appears in multiple seeds. Since a mapping has been defined as a bijective function, a valid multi-attribute mapping can include only one seed out of each cluster of seeds with a common attribute. We implicitly denote as mappings only the valid ones, while the invalid ones are automatically filtered out.

Considering the example tables $R(X)$ and $S(Y)$ in Figure 3.5, the mappings inserted into Seeds are the ones depicted in Figure 3.6. Here the mappings are represented by the attribute M (in this example we use the headers instead of the indices to enhance readability) and their area by the attribute A. Thus, the first seed, which maps the City attribute in R to the City attribute in S, has an area of 8 cells, since all cells from the first attribute find a match in the second one. The mappings involving the Team and Stadium pairs have smaller overlaps instead, due to the absence of Arsenal and its stadium from S and the use of two distinct denominations for the stadium of Inter Milan. Note that three teams (Milan, Liverpool, and Barcelona) carry the name of their city, causing the mappings from Team to City and from City to Team to be detected as seeds (denoted in Figure 3.6 as S_3 and S_4, respectively). By definition, S_1 and S_3 cannot occur together, since Team cannot be mapped to both Team and City (same for S_0 and S_3, S_0 and S_4, S_1 and S_4).

As stated above, the candidates can be seen as the nodes of the lattice representing the combinations of the seeds (depicted in Figure 3.6), where every level n contains the mappings obtained by combining n seeds. Since the lattice includes only the valid mappings, it is allowed to present an incomplete shape (hence we can define it as a *semilattice* in case an attribute appears



Barcelona	Barcelona	Camp Nou
Bayern Munich	Munich	Allianz Arena
Liverpool	Liverpool	Anfield
Manchester United	Manchester	Old Trafford
Milan	Milan	San Siro
Real Madrid	Madrid	Santiago Bernabéu

Figure 3.7: *Levels* as initialized (a) and after the update (b), *Candidates* (c), and the detected largest overlap (d).

in more than one seed). The highest level containing at least one valid mapping can be identified considering the minimum number of distinct attributes from a table covered by the seeds, i.e., $min(|X_{Seeds}|, |Y_{Seeds}|)$. When every attribute appears in at most one seed, the complete lattice is composed of a number of nodes equal to:

$$\sum_{n=1}^{s} \binom{s}{n} = 2^s - 1$$

In the lattice, for every node are reported the upper bounds for the height (between round brackets) and the area, where the latter is obtained as the product between the bounded height and the number of seeds combined to generate the candidate (i.e., $|X_M|$), representing the width of the overlap. These values are determined by the seeds. In fact, considering a candidate M, the maximum possible value for its height is determined by the seed with the smallest area appearing in the combination, since $|R[X_M] ⩀ S[M(X_M)]| \leq min(\{|R[x] ⩀ S[M(x)]|, \forall x \in X_M\})$. Therefore, we can say that a seed *dominates* the seeds with a greater area when they appear together in a candidate. Before being returned, the seed list is sorted by increasing dominance (Line 8), i.e., $Seeds[i].A \geq Seeds[i+1].A, 0 \leq i < s-1$. In the last position we find the seed with the minimum area, which dominates all other seeds. Note that the nodes in the bottom level of the lattice

Algorithm 3.3: initLevels() function

```
Input: The sorted list of the detected seeds
Output: The priority queue for the lattice levels
1 $Levels \leftarrow maxHeap(\emptyset, key = A)$
2 for $n \leftarrow 1$ to $min(|X_{Seeds}|, |Y_{Seeds}|)$ do
3     $level.w \leftarrow n$                                   // width
4     $level.i \leftarrow n - 1$                               // seed pointer
5     $level.A \leftarrow level.w \cdot Seeds[level.i].A$      // max area
6     if $level.A \geq \theta$ then
7         $Levels.push(level)$
8 return $Levels$
```

($n = 1$), representing the seeds themselves, already report the actual values for the height and the area.

In the example, the seeds cover three distinct attributes from both tables, hence the top level of the lattice in Figure 3.6 is the one containing the combinations of three seeds. Let us consider the candidate combining S_0 and S_1 (i.e., the first node of level 2*), with an area of 8 and 7 cells, respectively. While the width of its overlap is equal to 2, its height can be at most equal to 7, hence its area to 14.*

Prioritizing the Levels

Based on the number of detected seeds, the lattice can become significantly large. The possibility to define for every level an upper bound for the area of its candidates allows to generate the candidates incrementally according to their potential area, avoiding the materialization of the entire lattice. In fact, all candidates in a level n have a fixed width of n, and their height is bounded by the area of the seed in the position $n - 1$ of the sorted *Seeds* list (i.e., the first when $n = 1$, the second when $n = 2$, etc.), since in a level the first $n - 1$ seeds are always dominated by other seeds if they occur in a candidate. Thus, we can insert representations of the levels into *Levels*, the first of the two priority queues introduced above, to determine which level can produce the candidate with the maximum potential area and which seeds have to be combined for generating it (i.e., the seed bounding the height and the ones that precede it in the list).

The initialization of *Levels* is delegated to the **initLevels**() function (Function 3.3). This function initializes the *Levels* priority queue as an empty *max heap* structure (Line 1), then iterates over all possible levels the lattice may contain, starting from the bottom (i.e., level 1 that contains the

Algorithm 3.4: genCand() function

```
Input: The priority queues for the lattice levels and for the candidates,
       the sorted list of the detected seeds
Output: The updated priority queues
1  $topL \leftarrow Levels.pop()$                                  // top level
2  forall $comb$ in $combs(Seeds[: topL.i], topL.w)$ do
3      $cand.M = comb$                                            // mapping
4      $cand.w = topL.w$                                          // width
5      $cand.h = Seeds[topL.i].A$                                 // max height
6      $cand.A = cand.w \cdot cand.h$                             // max area
7      $cand.O \leftarrow \emptyset$                              // overlap
8      if $cand.A \geq \theta$ then
9          $Candidates.push(cand)$
10 if $topL.i < len(Seeds) - 1$ then
11     $topL.i \leftarrow topL.i + 1$                             // next seed
12     $topL.A = topL.w \cdot Seeds[topL.i].A$
13     if $topL.A \geq \theta$ then
14         $Levels.push(topL)$                                    // reinsert level
15 return $Levels, Candidates$
```

seeds), until it reaches the top of the lattice, which is limited by the minimum number of attributes covered by the seeds of either table, as stated above. For every level, the queue maintains the width of the candidates in that level (w) and a pointer i to the seed bounding their height, which consequently determines the upper bound for their area A (Lines 3-5). If this upper bound is equal to or greater than the pruning threshold θ, the level is inserted into *Levels* (Lines 6-7). Then, we can proceed with the upper levels.

In the example, the Levels priority queue is initialized with three elements (Figure 3.7a), one for each level of the lattice.

Generating the Candidates

The *Candidates* priority queue is initialized as an empty *max heap* structure in Line 5 of Algorithm 3.1 to be filled incrementally with the generated candidates. At the beginning of an iteration, we might need to materialize further candidates to ensure that at least one of the candidates with the maximum potential area has been generated and inserted there. The process for generating the candidates is described by the **genCand**() function (Function 3.4) and is depicted in Figure 3.7.

Considering the level n located at the top of *Levels* (Line 1), we generate the candidates corresponding to all combinations of n seeds composed of the seed pointed by the level and $n-1$ seeds selected among the ones preceding it in the list (Line 2). Note that this incremental process covers all candidates in the level, since:

$$\sum_{i=n-1}^{s-1} \binom{i}{n-1} = \binom{s}{n}, n \neq 0$$

and generates them in the correct ordering based on their potential area, since $Seeds[i].A \cdot n \geq Seeds[i+1].A \cdot n, n-1 \leq i < s-1$.

In the example, the candidate with the maximum potential area is generated from level 3*, located at the top of Levels. This level is currently pointing to the seed S_2, which occupies the third position in the seed list, preceded by S_0 and S_1. Since only three seeds are considered, only the first node of the third level of the lattice in Figure 3.6 needs to be materialized.*

For every generated candidate, we keep track of its width w (equal to n) and the upper bound for its height h, which determines its potential area A (Lines 4-6). The attribute O is initialized to be empty (Line 7) and is updated with the actual overlap when it is computed. Thus, when a candidate has the maximum priority, if its overlap is empty, we can infer that its attribute A describes its potential area, hence we still need to detect the overlap to verify its actual value. If the candidate is a seed, it is possible to directly initialize O with its actual overlap, which has already been computed. Finally, the candidate is inserted into the *Candidates* priority queue if its area is at least equal to θ or exceeds it (Lines 8-9).

In the example, θ is still equal to 8, reflecting the maximum area among the seeds; thus, the generated candidate is inserted into Candidates, which was previously empty.

After the new candidates have been generated, the considered level has to be updated, pointing now to the next seed in the list (unless we were already pointing to the last one) and updating the upper bound for the area of the candidates yet to be generated consequently (Lines 10-12). If the updated upper bound is greater than θ, then the level is reinserted into *Levels* (Lines 13-14).

This update step is represented for the example in Figure 3.7b; from S_2, level 3 points now to S_3, which sets the upper bound for the height to 3 and for the potential area to 9 (which is greater than θ, so the level can be reinserted, but now level 2 can produce candidates with a greater potential area). If the reinserted level was located at the top of Levels in one of the next iterations, it would have to generate the candidates corresponding to all combinations containing S_3 (the pointed seed) and two seeds among S_0, S_1, and S_2, producing in principle three new candidates (in this case, they would

Algorithm 3.5: verCand() function

```
Input: The two tables $R(X)$ and $S(Y)$, the top candidate, the priority
       queues for the lattice levels and for the candidates
Output: The updated priority queues
1  $topC.O \leftarrow R[X_{topC.M}] \Cap S[Y_{topC.M}]$            // overlap
2  $topC.h \leftarrow |topC.O|$                                     // actual height
3  $topC.A \leftarrow topC.w \cdot topC.h$                          // actual area
4  if $topC.A \geq \theta$ then
5      $\theta \leftarrow topC.A$
6      $Candidates.push(topC)$                                      // reinsert candidate
7      forall $cand$ in $Candidates$ do
8          if $cand.A < \theta$ then
9              $Candidates.delete(cand)$                            // prune candidate
10     forall $level$ in $Levels$ do
11         if $level.A < \theta$ then
12             $Levels.delete(level)$                               // prune level
13 return $Levels, Candidates$
```

be automatically filtered out, since they would all contain at least one seed between S_0 and S_1, not compatible with S_3).

Verifying the Candidates

If the actual overlap of the candidate located at the top of *Candidates* has not been verified yet, we need to call the **verCand**() function (Function 3.5).

First, we need to compute the overlap according to the definitions provided in Section 3.4 to obtain the actual values for its height and its area (Lines 1-3). If the actual area is at least equal to the pruning threshold θ, the candidate is reinserted into *Candidates* with the updated values, since it might be the largest overlap. Moreover, we update θ to ensure it reflects the value of the maximum actual area that we know at the moment (Lines 4-6). This dynamic updating allows to prune the lattice in a more efficient way. In particular, when θ is updated, we can scan both priority queues to delete the elements that cannot reach its new value (Lines 7-12).

In the example, verifying the top candidate (Figure 3.7c) produces the overlap depicted in Figure 3.7d, presenting an area of 18 cells. Since its area is greater than θ, the candidate is reinserted into Candidates and θ is set to 18, causing the pruning of all elements in Levels (Figure 3.7b), since they cannot produce candidates whose area can reach this threshold. At the next *iteration, the top candidate has already been verified and is therefore added to the result set, completing the task after one single verification.*

Coherently with the concept of dominance among seeds, since the height of a mapping is bounded by the seed with the smallest area appearing in the combination, adding further seeds to a mapping cannot increase this bound. Thus, we can state that the height of a node in the level n of the lattice is bounded by the height of the nodes from the level $n-1$ that are combined to generate it (we can equivalently say that they are *covered* by that node).

As a further optimization, when we compute the overlap of a candidate we can check if some elements in *Candidates* cover it and in this case bound their height based on the one of the verified candidate, i.e., $cand.h = min(topC.h, cand.h)$. Similarly, if we cache all verified heights with the related mappings, when we generate a candidate we can check if it covers some of those mappings and update its potential height consequently. These optimizations, overlooked in Algorithm 3.1 to improve readability but included in its implementation, make the potential area of the candidates closer to the actual one, enhancing the effectiveness of the pruning and the definition of the top candidate in the priority queue.

Computational Complexity

Let us consider tables T_1 and T_2, with r_1 and r_2 rows and c_1 and c_2 columns, respectively. For detecting the seeds, we need to consider all column combinations and compute their bag intersection. Using a hashmap, the cost is in $\mathcal{O}(r_1 * c_1 + r_2 * c_2 + c_1 * c_2 * \min(r_1, r_2))$: $\min(r_1, r_2)$ for the intersection itself and $r_{1(2)} * c_{1(2)}$ for the hashmap generation. The s detected seeds, $0 \leq s \leq \min(c_1, c_2)$, generate a lattice composed of $2^s - 1$ candidates, net of invalid mappings. In principle, considering the actual area A^* of the largest overlap, not known a priori, our algorithm needs to generate and verify all g candidates with a potential area $A \geq A^*$ (note that the additional optimizations introduced at the end of the previous section can significantly reduce the number of candidates to verify). These candidates are generated from Σ *generating seeds*, i.e., the total number of seeds pointed by the items of the *Levels* priority queue, at most $s - l + 1$ for each level l. Hence, Σ generating seeds produce a number of candidates g equal to:

$$g = \sum_{\sigma \in \Sigma} \binom{i_\sigma}{l_\sigma - 1}$$

where l_σ denotes the level pointing the seed and i_σ its index in *Seeds*, $0 \leq i_\sigma \leq s - 1$. In the worst case (i.e., if all seeds have the same area but no cell alignment), our algorithm might need to generate and verify through the bag intersection all candidates in the lattice (apart from the s seeds, whose actual area is already known), for a cost of:

$$\mathcal{O}(\sum_{l=2}^{s}\binom{s}{l} * l * (r_1 + r_2)) = \mathcal{O}(2^s * s * (r_1 + r_2))$$

Hence, the computational complexity of the algorithm is in practice dominated by the exponential complexity in the number of seeds.

3.5.2 Greedy Algorithm

Algorithm 3.1 for the detection of the largest overlap between two tables, described in Section 3.5.1, generates the candidates by combining the seeds. If we need to apply the algorithm to a pair of wide tables with some values repeated across several columns in both (e.g., multiple columns containing Boolean values), it is possible that a significant number of seeds is detected, producing a huge lattice mostly composed of invalid nodes. In some cases, this situation can lead to the impossibility of generating the combinations for the new candidates in a reasonable amount of time.

While in principle it would be possible to reduce the number of seeds by post-processing them (e.g., to make an attribute appear in at most k seeds or reducing this situation to the *stable matching* problem [Gale and Shapley, 1962], with the area of a seed determining its weight), pruning them a priori without knowing the actual area of any of the candidates in the upper levels can have a negative impact on the correct detection of the largest overlap.

For a result as close as possible to the exact largest overlap, we designed a greedy variant for our algorithm inspired by *beam search*, a heuristic search algorithm widely applied in the speech recognition area [Huang et al., 2014] that performs a breadth-first search in a tree by only expanding the β most promising nodes at each level. The parameter β, denoted as *beam width*, is defined by the user based on the trade-off between efficiency (a smaller value for β requires to evaluate fewer candidates) and completeness (the case for $\beta = \infty$ would evaluate all candidates, producing the exact result).

Our greedy algorithm is designed to bottom-up traverse the lattice generated from the seeds. The set of the largest overlaps is initialized using the seeds with the maximum area (if it is greater than the minimum area Δ defined by the user). This value is used to initialize the pruning threshold θ, which can be updated (as well as the result set) every time we verify a new candidate. Before moving to the upper levels, we check if it is still possible for any remaining candidate to reach (or even surpass) θ. This is possible because the combination of bounded heights and width determines a bound for the area of their overlaps. If the current θ is larger than all such upper bounds of candidate areas, the algorithm saves on further exact area computations and terminates.

At the beginning, we consider the detected seeds and select a maximum of β candidates with the greatest area. To find candidates for the second level, we combine each of them with every other seed and drop repeated and invalid combinations. After this generating step, we verify all new candidates and again select only the β candidates with the greatest actual area among them. For the third and every further level, we combine the selected candidates of the previous level with every seed that is not already part of the candidate and then again verify their area and limit their number to β. Note that the selection of the best β candidates may affect the produced results; thus, it is important to rely on some tiebreaker strategies for the candidates that present the same actual area (e.g., favoring the ones whose seeds present the greatest total area).

To determine the computational complexity, let us again consider tables T_1 and T_2, with r_1 and r_2 rows and c_1 and c_2 columns, respectively. Since from level 2 up we generate the candidates to verify by combining the top β ones from the previous level with all s seeds (net of generated duplicate candidates and seeds that would raise invalid ones), the cost of our greedy algorithm can be quantified as:

$$\mathcal{O}(\sum_{l=2}^{s} \beta * s * l * (r_1 + r_2)) = \mathcal{O}(s^3 * \beta * (r_1 + r_2))$$

Combined with the (unchanged) initial seed detection, this leads to an overall computational complexity of $\mathcal{O}(r_1 * c_1 + r_2 * c_2 + c_1 * c_2 * \min(r_1, r_2) + s^3 * \beta * (r_1 + r_2))$, i.e., polynomial in the number of seeds. Our experimental evaluation (Section 3.6.2) demonstrates that, even in absence of approximation guarantees on the quality of its result, the greedy algorithm is generally able to detect largest overlaps of the same area as those discovered by the exact algorithm.

Coordinating the Algorithms

As described at the beginning of Section 3.5, SLOTH adopts a *trial-and-error* approach to evaluate a pair of tables, favoring the exact algorithm whenever possible. The main reasons behind this choice are: *(i)* the guarantees of correctness and completeness of the result given by the exact algorithm (not guaranteed by its greedy variant, despite the empirical demonstration of its good accuracy); *(ii)* the minimal impact of timeouts on our experiments on coexisting tables from real-world scenarios (Sections 3.6.3 to 3.6.5), where only around 1.6% of all table pairs need to revert to the greedy algorithm. Further, SLOTH can be seen as a unique *best-effort* solution, since several findings by the exact algorithm, such as the detected seeds and the computed actual heights and overlaps, can be reused by its greedy variant.

The main factor leading to timeouts is the number of seeds, which directly affects the size of the lattice, hence the number of candidates that potentially need to be generated. This aspect is strictly correlated to the width of the tables and the amount of repeated cell values across them (Figures 3.8a and 3.8b). Nevertheless, as depicted in Figures 3.8c and 3.8d, none of these factors defines a clear threshold to distinguish between successes and timeouts. The definition of rules for the automatic selection of the algorithm is therefore not trivial, while a classifier-based approach, able to detect more complex patterns among the described features, might seem more promising. However, to pursue such an approach, several factors need to be taken into account. First, training data is needed, and, as stated above, in several table corpora timeouts are rather rare. Further, tables from different corpora might have very dissimilar features, as shown in Table 3.1, hence training the classifier on one dataset (e.g., the sample of wiki_history described in Section 3.6.1, where timeouts are quite frequent) might not cover many patterns occurring in other datasets, causing a poor accuracy. False positives might not only miss the possibility of detecting the exact result, but also impact negatively on the performance. As shown in Figure 3.8i, in many cases the exact algorithm can be faster than its greedy variant, since it can directly prioritize the evaluation of the most promising candidates from the lattice, while the latter proceeds level by level from bottom to top.

3.6 Experimental Evaluation

This section reports the experimental evaluation of SLOTH, aiming to assess: *(i)* what is the performance of SLOTH at detecting the largest overlap between two tables based on the size and the features of those tables (Section 3.6.1); *(ii)* what is the accuracy of the results obtained using the greedy algorithm (Section 3.6.2); *(iii)* how many highly overlapping tables are present in a real-world relevant scenario such as Wikipedia and what we can learn from their analysis (Section 3.6.3); *(iv)* how SLOTH can be applied to real-world use cases such as the detection of potential copying between tables from different sources (Section 3.6.4) or the detection of composite foreign keys in the relational context (Section 3.6.5).

Table 3.1: Statistics about the number and the size of the tables appearing in the used datasets.

Dataset	#D	Width (#columns)			Height (#rows)		
		MIN	MAX	AVG	MIN	MAX	AVG
wiki_history	55.97M	1	5694	5.92	1	17.38k	26.63
wiki_latest	2.13M	1	883	5.23	1	4670	11.47
uni_dwh	158	1	55	9.48	2	151.78k	5604.79
stock_raw	1.15k	4	69	16.18	221	1000	987.58
stock_clean	1.15k	3	17	12.49	221	1000	987.58
flight_clean	1.17k	3	7	5.75	6	1309	662.17

Datasets

Table 3.1 presents the datasets used in our experimental evaluation. We denote as wiki_history the Wikipedia table matching dataset from the IANVS project[7]. Differently from other corpora of Wikipedia tables, such as WikiTables[8] [Bhagavatula et al., 2015], composed of 1.6M high-quality relational tables, this dataset captures the evolution of all 3.5M tables present in the English Wikipedia throughout its entire history (until September 1, 2019), for a total amount of 55.97M different table versions stored in the JSON Lines text format [Bleifuß et al., 2021b]. The composition of the *table lineages* (i.e., the collections of the subsequent versions of a table) was performed by Bleifuß et al. in their structured object matching project [Bleifuß et al., 2021a]. We separately consider the most recent snapshot of Wikipedia tables from this dataset, denoted as wiki_latest.

Beyond the Wikipedia scenario, we also employ uni_dwh [Castro Fernandez et al., 2018], a real-world university data warehouse, composed of relational tables with a significantly higher number of rows, and two datasets[9] [X. Li et al., 2012] reporting the information about stock symbols and flights captured from different sources across multiple days (55 sources over 21 days and 38 over 31, respectively); thus, on every day there is a table for each source. For the stock dataset both the original tables (stock_raw) and their versions obtained through schema alignment (stock_clean) are available, while only the latter is provided for the flight dataset.

[7]`https://hpi.de/naumann/projects/data-profiling-and-analytics/change-exploration.html`

[8]`http://websail-fe.cs.northwestern.edu/TabEL/`

[9]`https://lunadong.com/fusionDataSets.htm`

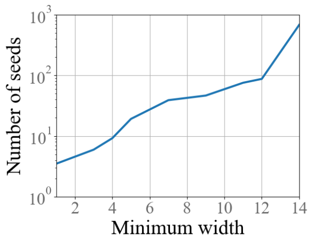
Setup

Sloth has been implemented in Python 3.7. We used a MongoDB instance to store the tables and their metadata. Our experiments were performed on a server machine equipped with 4 Intel Xeon E5-2697 @ 2.40 GHz (72 cores) processors and 216 GB of RAM, running Ubuntu 18.04. The default configuration for Sloth adopts a timeout of 3 seconds for the exact algorithm and 60 seconds for the greedy algorithm (with a default beam width equal to 32).

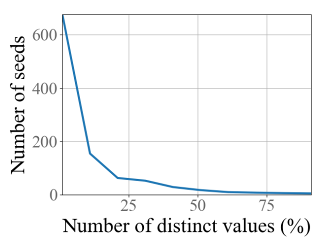
3.6.1 Performance of the Algorithms

The performance of Sloth has been evaluated on two of the datasets presented above: a representative subset of the wiki_history dataset and the uni_dwh dataset, chosen to cover both the case of Web tables and the one of relational tables collected from a database in our analysis. In particular, the subset was created by randomly picking from wiki_history at most 5 tables from each table lineage (to include in the evaluation also these cases of highly related tables) and filtering out the ones with less than 10 rows, for a total of 4.1M tables. We then produced 19.1M pairs of tables through LSH banding [Leskovec et al., 2020], using 16 bands and a minhash of 128 bits, and randomly selected 1M of these pairs to be used in our evaluation. Our results are reported in Figure 3.8, explained in detail in the following.

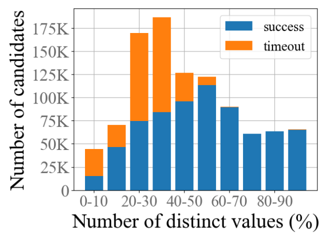
First, we examine the factors that determine the number of seeds and thus the size of the lattice. Figure 3.8a shows how the number of seeds increases with the number of columns, as expected. For table pairs with different number of columns, we consider the smaller number, which also bounds the height of the lattice. We plot the average number of seeds for 10 quantiles, positioning the values at the beginning of the interval covered by the quantile on the x-axis. A second dimension that has a high impact on the number of seeds is the percentage of distinct cell values in the two tables. Given a pair of tables, we consider the two sets of cell values, then divide the sum of their sizes by the total number of cells of the two tables. In Figure 3.8b, aggregating this value into 10% buckets, we show that the number of seeds tends to significantly increase when the tables contain many repeated values. These are also the most challenging pairs for Sloth, as the distribution of the cases for which the exact algorithm exceeds the timeout shows in Figure 3.8c. In fact, these 291k pairs fall almost entirely in the initial half of the graph and mostly reflect the scenario in which two tables contain the repetition of few distinct cell values in several different alignments, producing many seeds and requiring our exact algorithm to generate many candidates before



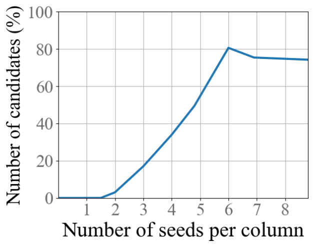
(a) Seeds per minimum table width (wiki).



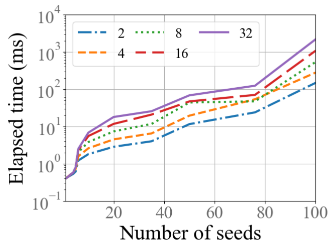
(b) Seeds per distinct cell values (wiki, exact).



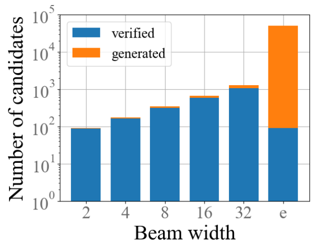
(c) Timeouts per distinct cell values (wiki, exact).



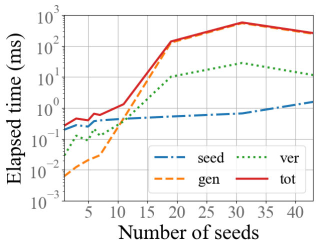
(d) Timeouts per number of seeds (wiki, exact).



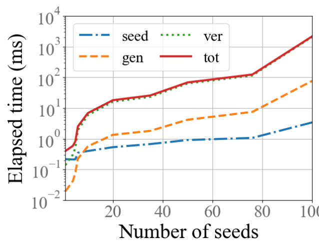
(e) Total time per beam width (wiki, greedy).



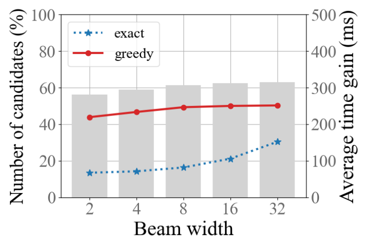
(f) Materialized candidates per beam width (wiki).



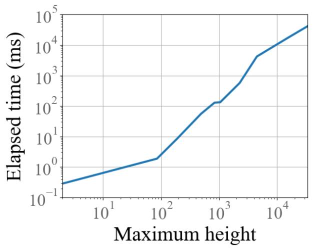
(g) Separated time for each task (wiki, exact).



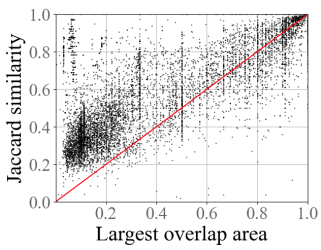
(h) Separated time for each task (wiki, greedy, $\beta = 32$).



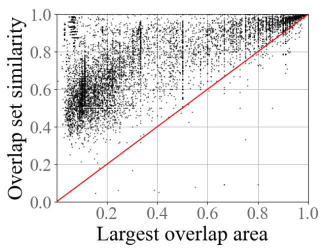
(i) Candidates where exact is faster than greedy (wiki).



(j) Time for seed detection per maximum table height (dwh).



(k) Largest overlap vs. Jaccard similarity (wiki).



(l) Largest overlap vs. overlap set similarity (wiki).

Figure 3.8: Performance of SLOTH evaluated on the wiki_history dataset (a-i, k-l) and the uni_dwh dataset (j).

discovering the largest overlap. In Figure 3.8d, we equivalently show how the percentage of timeouts increases with the average number of seeds per column (computed on the table with the smaller width, always considering 10 quantiles).

In case of timeout, SLOTH activates the greedy algorithm based on beam search. The impact of the beam width is illustrated for 10 quantiles in Figure 3.8e. Using a larger beam width tends to produce more reliable results (see Section 3.6.2), but it also implies materializing more candidates, whose area needs to be verified to select the most promising ones, as highlighted by Figure 3.8f, hence requiring more time. Thus, the candidate verification represents by far the most expensive operation for the greedy algorithm, as illustrated in Figure 3.8h, which analyzes the time (once again as the average value for 10 quantiles) required by the three main tasks performed by SLOTH (the other two being seed detection and candidate generation). For our exact algorithm, the most critical task is the candidate generation instead, as highlighted by the last column of Figure 3.8f and by Figure 3.8g, while the use of the priority queues allows minimizing the number of candidates whose actual area needs to be verified out of the generated ones. In Figure 3.8i, the bars show for each beam width the percentage of candidates for which the exact algorithm (when it does not exceed the timeout) is faster than its greedy variant. The lines show for each algorithm the average time gain on the candidates for which it is the fastest.

Finally, the number of rows can affect those tasks that require to perform the bag intersection, in particular the seed detection. While the effect is very limited in the case of Wikipedia, it becomes much more evident when moving to the relational tables of uni_dwh, as presented in Figure 3.8j, where the average number of rows is significantly larger (see Table 3.1) and the time required for detecting the seeds can exceed 5 minutes in some extreme cases.

3.6.2 Accuracy of the Greedy Algorithm

Table 3.2 reports the accuracy of the largest overlaps detected by the greedy algorithm, computed on the 1M random table pairs introduced in Section 3.6.1. In particular, we consider five different values for the beam width (β) and we report for each of them: *(i)* the percentage of pairs for which the computation exceeds the timeout; *(ii)* the percentage of pairs for which the ratio between the area of their largest overlap and the one obtained by the exact algorithm is equal to 1 (i.e., same area) or at least 0.75 (i.e., close to the exact area), computed on the pairs where the exact algorithm provides a solution (708.2k) and its greedy variant does not exceed the timeout; *(iii)* the same percentage considering the ratio w.r.t.

Table 3.2: Accuracy of the greedy algorithm results with different beam width (β) values, compared to the exact result and the largest greedy result as the ratio between their areas.

β	**Timeout**	**Accuracy (exact)**		**Accuracy (greedy)**	
		1	**$\geq$ 0.75**	**1**	**$\geq$ 0.75**
2	0.040%	98.612%	99.668%	72.988%	96.707%
4	0.041%	99.351%	99.857%	80.940%	98.718%
8	0.042%	99.835%	99.992%	87.326%	99.392%
16	0.046%	99.917%	99.998%	92.025%	99.700%
32	0.127%	99.940%	99.999%	96.311%	99.915%

the largest overlap with the maximum area among those produced by the different beam width values (also considering the case with $\beta = 64$), to study the accuracy of the greedy algorithm on the pairs where the exact algorithm exceeds the timeout.

As depicted in Table 3.2, in the first case (third and fourth columns) even smaller values for the beam width lead to a very high accuracy. Differently, in the second one (last two columns), they struggle to reproduce the exact result (despite producing a good approximation); hence, values such as 16 or 32 appears to be a more reliable choice (note that these values reach a very high accuracy of at least 0.9 on 97.148% and 99.013% of the pairs, respectively), even if these configurations require more computational time, as highlighted by their increasing (yet marginal) timeout rate.

3.6.3 Overlapping Tables in Wikipedia

With SLOTH, we are able to detect overlapping tables coexisting in the latest snapshot of the Wikipedia table matching dataset, composed of 2.13M tables (see Table 3.1). To avoid comparing all pairs, we first performed LSH banding on the dataset, using 8 bands and a minhash of 128 bits[10]. This operation reduced the candidate set to 6.91M pairs of tables (we ignored the trivial tables with only one distinct cell value).

Among the remaining candidates, the greedy algorithm was required only for 110.88k of them (1.605% of the cases), and for only 4 the timeout of 60 seconds was exceeded. The average time for evaluating a candidate was 153.7 ms, and the process could be easily parallelized, distributing the candidates over several workers.

[10]This approximates a Jaccard similarity threshold of about 0.8; we observed in preliminary experiments, by manually inspecting samples of candidates, that under that threshold the table overlap occurs mostly by chance in this dataset.

Table 3.3: Types of overlapping tables in Wikipedia.

Type	**#Table Pairs**	**Estimated Size**
Perfect duplicates	5.91M (85.521%)	39.55 MB
Inclusions	351.24k (5.085%)	6.05 MB
Additional rows	59.75k (0.865%)	4.35 MB
Additional columns	289.97k (4.198%)	1.91 MB
Additional rows and columns	1.53k (0.022%)	0.56 MB
Partial overlaps	648.91k (9.394%)	39.16 MB
$\geq$ *50% of the smallest table*	252.80k (3.660%)	35.52 MB
$<$ *50% of the smallest table*	396.12k (5.735%)	6.60 MB
Total ($\geq$ 50%)	**6.51M (94.265%)**	**63.49 MB**

Table 3.3 reports the results of our analysis, categorizing the evaluated candidates according to the following types of overlap: *(i) perfect duplicates*, if the two tables have identical content (with the possible reordering of columns and rows); *(ii) inclusions*, if the smaller table is contained in the larger one, which presents some additional rows and/or columns; *(iii) partial overlaps*, if both tables present some cells that are excluded from the largest overlap. For the estimated memory occupation, we rely on the average size of the cells in this snapshot, equal to 14.53 bytes.

The number of pairs with an overlap of at least 50% of the smaller table is surprisingly high. In particular, Wikipedia contains a huge amount of perfect duplicates; this situation denotes a wide diffusion of the copy-and-paste practice across multiple pages, which can foster inconsistencies, as highlighted by the relevant presence of inclusions and partial overlaps (e.g., we notice how even the frequent and apparently trivial case of the legend tables defined for a certain category of pages can lead to the rise of differences during the evolution of these tables), and requires a consistent editing effort by the users to achieve coherency in the encyclopedia.

Finally, we show how the area of the largest overlap detected by SLOTH, normalized by the area of the smaller table, differs from traditional metrics based on set semantics, such as Jaccard similarity [Jaccard, 1912] and overlap set similarity [D. Deng et al., 2018; E. Zhu et al., 2019]. Set semantics cannot consider the repetition of cell values and their alignment in the table; moreover, Jaccard similarity presents a bias against sets of different sizes. Figures 3.8k and 3.8l show on a random sample of 10k pairs (from the 1M random table pairs introduced in Section 3.6.1) how these substantial differences can lead Jaccard and overlap set similarity (normalized by the size of the smaller set) to very dissimilar results from SLOTH. For instance, with a 0.8 threshold SLOTH would detect 321k pairs out of 1M, while Jaccard

Table 3.4: Size of the clusters of sources with potential copying.

Dataset	**Sloth** *(min_height)*	**Jaccard** *(threshold)*	**Li et al.**
stock_raw	13, 2 *(0.85)*	12, 2 *(0.80)*	11, 2
stock_clean	13, 2 *(0.95)*	12, 2 *(0.85)*	
flight_clean	5, 4, 3, 3, 3 *(0.95)*	5, 4, 3, 2, 2 *(0.80)*	5, 4, 3, 2, 2

and overlap set similarity 339k and 557k (268k and 318k in common with Sloth). For 365k and 601k pairs, Sloth's result differs by at least 0.2 from Jaccard and overlap set similarity, while the unnormalized value for the latter is more than double or less than half the largest overlap area in 334k cases. We collected in our GitHub[11] repository 50 representative example pairs from the wiki_history dataset depicting typical cases where the analyzed metrics differ significantly.

3.6.4 Potential Copying Detection

As a further real-world use case, in this section we focus on the detection of potential copying across different sources. In their work on truth finding [X. Li et al., 2012], Li et al. detect clusters of sources with potential copying on the datasets about stocks and flights introduced in Table 3.1, relying on criteria such as claimed dependencies or query redirections, and computing several measures on them. The right column of Table 3.4 shows the size of those clusters.

To perform potential copying detection with Sloth, we configured the algorithm with a high threshold for the minimum height of the largest overlap to be detected (e.g., 0.9 of the table with fewer rows in the pair) and a minimum width of 2 (since all tables share the column with the object identifiers). This way, the algorithm finds pairs of tables with a very large overlap on a subset of columns. Note that we consider the stock dataset both in the raw and in the clean version that is obtained through schema alignment. Due to the limited size of the datasets and the early stopping introduced by the defined bounds, we simply consider for each day all the pairs of tables obtained through the Cartesian product, net of identity and reflexivity. Sloth finishes all computations for each single day in less than one minute. Finally, we consider those pairs of tables with an overlap width of at least 0.8 of the table with fewer columns (considering the lower bound for the schema similarity of the clusters detected by Li et al.) during the whole period, with a tolerance of 3 days, to be potential copies. As a baseline, we adopt Jaccard

[11]`https://github.com/dbmodena/sloth/tree/main/examples`

similarity (on which Li et al. rely for multiple measures), computed between the sets of cell values of the tables.

As depicted in Table 3.4, both solutions are not only able to detect all clusters found by Li et al., but also to retrieve some additional sources with potential copying in both domains. While a source with almost exactly the same schema and content is easily detected even by Jaccard similarity as a part of the larger cluster of the stock domain, SLOTH can retrieve three more additional sources: first, a second source for the same cluster with a significantly wider schema and different labels (which can be retrieved by Jaccard similarity only when dropping its additional columns in the aligned version, with a threshold of 0.75); second, two more sources in the flight domain, composed of a limited amount of copied rows. Indeed, Jaccard similarity struggles with sets of different sizes; its set semantics, ignoring the repetition and the alignment of cell values, might even lead to the detection of false positives. Thus, using SLOTH we were able to detect all clusters of sources with potential copying with a minimum effort, even without the need of performing schema alignment (as highlighted through the stock_raw dataset), also leading to the discovery of meaningful additional sources.

3.6.5 Discovery of Candidate Multi-Column Joins

SLOTH can support practitioners in the fundamental yet challenging task of automatically discovering candidate multi-column joins in a corpus of tables. To demonstrate it, we employ the real-world uni_dwh dataset, which is composed of 158 tables, making 12.4k table pairs. The average time for computing the largest overlap for each table pair is about 9.6 seconds (mainly due to the seed detection, as pointed out in Section 3.6.1, not counted towards the timeout) and the process could be easily parallelized.

To detect reasonable multi-column joins, we limit the largest overlap to between 2 and 5 columns. We also require the height of the overlap to be at least 90% of the table with fewer rows: this ensures high coverage of its values, while not looking for a perfect containment, which would be limiting for the join discovery scenario. For instance, say we find that two tables have a largest overlap involving 90% of the rows of one table and three attributes `FirstName`, `LastName`, and `Position`; then, we might use these attributes for joining the two tables.

A largest overlap that complies with the defined settings is detected for only 243 pairs (out of 5.6k pairs that would be obtained without defining restrictions on the overlap size). It is easy to determine the cardinality of the join by counting the duplicates in the original tables of the attributes that participate in the overlap: for each table in the pair, if there are no duplicates in the projection on those attributes, then the table participates in the join with cardinality one. Thus, among the detected overlaps, we find that 48, 64, 131 correspond to *one-to-one*, *one-to-many*, *many-to-many* joins, respectively. For 34 out of these pairs, the overlap detects a *composite key* (primary or alternate) for at least one table.

None of the existing solutions supports the automatic discovery of candidate multi-column joins. As a baseline, we can therefore adapt JOSIE, which uses the overlap set similarity to detect single-column joins, to consider as a set the entire tables instead of the single columns. Differently from SLOTH, such a baseline cannot take into account the structure of the table (including the repetition and the alignment of cell values), preventing the definition of the bounds that ensure the detection of multi-column joins. In fact, only 106 of the candidate multi-column joins discovered by SLOTH (48, 37, and 21 for each join type, 27 covering a composite key) are present among the top 243 table pairs according to their overlap set similarity (normalized by the size of the smaller set) retrieved by the baseline. Instead, the baseline detects several single-column joins and many invalid candidates, such as pairs with a low row coverage or where the values from a column are matched by multiple columns on the other side.

Chapter 4

Conclusion and Future Work

Data integration is the process of combining data acquired from multiple autonomous sources to provide users with a unified consistent view on this data. Together with data preparation and cleaning, it enables to guarantee the quality and enhance the value of the data at hand for its use in downstream tasks (e.g., to perform data analysis or to train artificial intelligence models). Data integration represents therefore one of the longstanding challenges in data management.

While in the past data integration had to deal with a relatively limited number of sources, the scenario has drastically changed with the explosion of big data. Nowadays, data integration is often required to work efficiently over millions of heterogeneous sources, such as databases, Web tables, open data, and data collected from sensors or IoT devices. This novel scenario poses several challenges to the historical data integration techniques and approaches, which have been significantly revised to comply with the features of big data.

In particular, the paradigm for data integration moved more and more from ETL (Extract, Transform, Load) towards ELT (Extract, Load, Transform). Indeed, cleaning the entire data to load it into a data warehouse would be prohibitively expensive and often technically unfeasible when dealing with big data. Thus, in ELT a huge amount of raw data is collected and directly stored as it is, for instance in a data lake, so that practitioners can transform and integrate useful portions of this large data corpus according to the task at hand.

Novel solutions are therefore required to handle only the portion of the data that is effectively needed for the task at hand, returning results in a timely manner, often in a pay-as-you-go fashion. This is what we denote as *data integration on-demand.*

Main Contributions

This thesis presented two main contributions that enrich the range of solutions to support data practitioners in the ELT paradigm. In particular, their focus is on two fundamental steps of the big data integration process: entity resolution (which aims to detect the records, inside a dataset or across multiple datasets, that refer to the same real-world entity, producing a consistent representation for it) and dataset discovery (whose goal is to retrieve related tables from large table corpora, e.g., to join them).

The first presented contribution is BrewER [Simonini et al., 2022; Zecchini et al., 2023; Simonini et al., 2023], a novel solution designed to perform *entity resolution on-demand*. BrewER enables data scientists and practitioners to run SQL SP queries directly on dirty data, obtaining the progressive emission of consistent results as if they were issued on the cleaned version of such data, according to a defined priority. BrewER is implemented as an open-source Python library, so that it can be seamlessly integrated by practitioners into their entity resolution workflows in Jupyter notebooks. As clarified through its experimental evaluation, BrewER can save a significant amount of time and resources in multiple relevant real-world scenarios.

The second contribution presented in this thesis is Sloth [Zecchini et al., 2024], a novel solution conceived to efficiently determine the *largest overlap* between two tables. Sloth aims to answer an open challenge in data discovery, the detection of *duplicate tables*, leading to several benefits both on the Web and in data lakes. For instance, it allows spotting and solving common data quality issues, such as inconsistent or incomplete information. Also, it helps eliminate redundancy to free up storage space or to save additional work for the editors, preventing the insurgence of data quality problems.

The experimental evaluation assesses the performance of Sloth in real-world scenarios, considering Web tables from Wikipedia and relational tables from a data warehouse, up to use cases such as the detection of potential copying between different sources and the automatic discovery of candidate multi-column joins in a corpus of relational tables.

Future Work

As well as being relevant contributions to the research about the respective big data integration steps, BrewER and Sloth also present interesting prospects for further developments.

Future Directions for BrewER

In the case of BrewER, one of the most impactful research contributions would be moving from SP to SPJ SQL queries, i.e., providing also support to join operations. As shown by some related previous work [Whang and Garcia-Molina, 2012], dealing for instance with *progressive relational entity resolution* [Altowim et al., 2014], this appears to be a challenging task. Nevertheless, proposing a solution to this problem would clearly represent a major advancement in the field.

Another relevant integration to the current implementation would be supporting *unbounded aggregate functions*, such as SUM. Enabling the application of this kind of functions to the attribute defining the emission priority (i.e., the one used in the ORDER BY clause) would require to revise the current algorithm, keeping track at each iteration of the maximum or minimum value (depending on the type of ordering, i.e., ASC/DESC) for each block of records, then using these *block bounds* to prioritize the evaluations of the records from the block that may obtain the best aggregate value in that moment. Similarly, the introduction of this additional block level can favor the parallelization of the computation, aimed to evaluate multiple blocks among the most promising ones at the same time, relying for instance on Apache Spark[1], as done for related techniques such as meta-blocking [Simonini et al., 2019; Gagliardelli et al., 2019].

Finally, an aspect closely related to the explainability of entity matching, described in Section 2.1, is the *fairness* of such matchers. Even if the research on fair entity resolution is still in its early stages, with only a few recent papers dealing with it [Shahbazi et al., 2023], it represents a significant and challenging direction to be explored [Azzalini et al., 2023]. An example of fair entity resolution approach is FairER [Efthymiou et al., 2021], a solution designed to guarantee the representativeness of a declared protected class when prioritizing comparisons in progressive entity resolution. BrewER may therefore be adapted to go significantly beyond this first approach, defining a protected class (or possibly even multiple classes) through a further additional clause in its SQL query, then guaranteeing the representativeness of this class among the top-k resulting entities. The usefulness of such a solution would go beyond strict fairness issues, guaranteeing for instance that a certain class of entities is represented in the obtained set of clean entities to be used in data analysis or for training a machine learning model on it.

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

Future Directions for Sloth

Moving instead to SLOTH, we plan to broaden our current research in multiple main directions. Firstly, we want to study the design of updatable indexes to enable table-overlap-based discovery at scale, i.e., to allow users to provide a table as a query and to retrieve the top-k tables presenting a large overlap with it. As stated in Section 3.3, some of the related solutions may be adapted for this task, for instance JOSIE [E. Zhu et al., 2019], considering a variant taking into account all table cells under the bag semantics to return the top-k most promising table pairs to evaluate. Beyond this opportunity, we plan to explore multiple alternative solutions, such as the use of XASH [Esmailoghli et al., 2022] or even novel indexing techniques specifically designed for this problem.

Another approach that would favor scalability is the generation of embeddings for the table cells, similarly to the approach adopted by EMBDI [Cappuzzo et al., 2020]. This would also allow to relax the cell matching operation, including in the overlap not only identical cells but also highly similar ones, based on the closeness of their embeddings.

A further relevant advancement would be the possibility to detect not only the largest, but also the *best* overlap between two tables, defining quality metrics to capture the meaningfulness of an overlap based for instance on its width and height or on the entropy of its content. This would also enable to mask inconsistencies with null values to avoid losing partially matching information, hence introducing a related penalty in the overlap quality [Batini et al., 2009].

Finally, considering the usefulness of SLOTH in the case of Wikipedia (as proven through our experimental evaluation), it would be worth creating a solution specialized to this scenario, reducing the search space and the number of accidental matches by further differentiating columns annotating them based on their type [Hulsebos et al., 2019] and considering even the context, expressed for instance by the page title (always available for every table), the section title, and the table description.

List of Publications

Publications covered by this thesis

- Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, Felix Naumann (2022). Entity Resolution On-Demand. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 7, pp. 1506–1518. https://doi.org/10.14778/3523210.3523226.

- Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi, Felix Naumann (2023). BrewER: Entity Resolution On-Demand. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 12, pp. 4026–4029. https://doi.org/10.14778/3611540.3611612.

- Giovanni Simonini, Luca Zecchini, Felix Naumann, Sonia Bergamaschi (2023). Entity Resolution On-Demand for Querying Dirty Datasets. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD).* CEUR Workshop Proceedings, vol. 3478, pp. 410–419. https://ceur-ws.org/Vol-3478/paper70.pdf.

- Luca Zecchini, Tobias Bleifuss, Giovanni Simonini, Sonia Bergamaschi, Felix Naumann (2024). Determining the Largest Overlap between Tables. *Proceedings of the ACM on Management of Data (PACMMOD)*, vol. 2, no. 1, art. 48. https://doi.org/10.1145/3639303.

Further relevant publications

- Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi (2020). Entity Resolution on Camera Records without Machine Learning. *Proceedings of the International Workshop on Challenges and Experiences from Data Integration to Knowledge Graphs (DI2KG @ VLDB).* CEUR Workshop Proceedings, vol. 2726. https://ceur-ws.org/Vol-2726/paper3.pdf.

- Luca Zecchini (2021). Progressive Query-Driven Entity Resolution. *Proceedings of the International Conference on Similarity Search and Applications (SISAP).* Lecture Notes in Computer Science, vol. 13058, pp. 395–401. *Awarded as Best Doctoral Symposium Paper.* `https://doi.org/10.1007/978-3-030-89657-7_30`.

- Luca Gagliardelli, Luca Zecchini, Domenico Beneventano, Giovanni Simonini, Sonia Bergamaschi, Mirko Orsini, Luca Magnotta, Emma Mescoli, Andrea Livaldi, Nicola Gessa, Piero De Sabbata, Gianluca D'Agosta, Fabrizio Paolucci, Fabio Moretti (2022). ECDP: A Big Data Platform for the Smart Monitoring of Local Energy Communities. *Proceedings of the EDBT/ICDT Workshops.* CEUR Workshop Proceedings, vol. 3135. `https://ceur-ws.org/Vol-3135/dataplat_short4.pdf`.

- Luca Zecchini (2022). Task-Driven Big Data Integration. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD).* CEUR Workshop Proceedings, vol. 3194, pp. 627–632. `https://ceur-ws.org/Vol-3194/paper75.pdf`.

- Luca Gagliardelli, Luca Zecchini, Luca Ferretti, Domenico Beneventano, Giovanni Simonini, Sonia Bergamaschi, Mirko Orsini, Luca Magnotta, Emma Mescoli, Andrea Livaldi, Nicola Gessa, Piero De Sabbata, Gianluca D'Agosta, Fabrizio Paolucci, Fabio Moretti (2023). A big data platform exploiting auditable tokenization to promote good practices inside local energy communities. *Future Generation Computer Systems (FGCS)*, vol. 141, pp. 595–610. `https://doi.org/10.1016/j.future.2022.12.007`.

- Andrea De Angelis, Maurizio Mazzei, Federico Piai, Paolo Merialdo, Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, Donatella Firmani, Xu Chu, Peng Li, Renzhi Wu (2023). Experiences and Lessons Learned from the SIGMOD Entity Resolution Programming Contests. *ACM SIGMOD Record*, vol. 52, no. 2, pp. 43–47. `https://doi.org/10.1145/3615952.3615965`.

- Angelo Mozzillo, Luca Zecchini, Luca Gagliardelli, Adeel Aslam, Sonia Bergamaschi, Giovanni Simonini (2023). Evaluation of Dataframe Libraries for Data Preparation on a Single Machine. *arXiv*, art. 2312.11122. *Currently under peer review at the IEEE International Conference on Data Engineering (ICDE).* `https://doi.org/10.48550/arXiv.2312.11122`.

- Adeel Aslam, Giovanni Simonini, Luca Gagliardelli, Luca Zecchini, Sonia Bergamaschi (2023). Stream-Aware Indexing for Distributed Inequality Join Processing. *Social Science Research Network (SSRN)*, art. 4424624. *Currently under peer review at Information Systems (IS).* `https://doi.org/10.2139/ssrn.4424624`.

Other publications

- Giovanni Simonini, Henrique Saccani, Luca Gagliardelli, Luca Zecchini, Domenico Beneventano, Sonia Bergamaschi (2021). The Case for Multi-task Active Learning Entity Resolution. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD).* CEUR Workshop Proceedings, vol. 2994, pp. 363–370. `https://ceur-ws.org/Vol-2994/paper40.pdf`

- Giovanni Simonini, Luca Gagliardelli, Michele Rinaldi, Luca Zecchini, Giulio De Sabbata, Adeel Aslam, Domenico Beneventano, Sonia Bergamaschi (2022). Progressive Entity Resolution with Node Embeddings. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD).* CEUR Workshop Proceedings, vol. 3194, pp. 52–60. `https://ceur-ws.org/Vol-3194/paper6.pdf`

- Donatella Firmani, Giovanni Simonini, Donatello Santoro, Jerin George Mathew, Luca Zecchini (2023). Bridging the Gap between Buyers and Sellers in Data Marketplaces with Personalized Datasets. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD).* CEUR Workshop Proceedings, vol. 3478, pp. 525–534. `https://ceur-ws.org/Vol-3478/paper01.pdf`

- Luca Gagliardelli, Domenico Beneventano, Marco Esposito, Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi, Fabio Miselli, Giuseppe Miano (2023). DXP: Billing Data Preparation for Big Data Analytics. *arXiv*, art. 2312.12902. `https://doi.org/10.48550/arXiv.2312.12902`.

Bibliography

Mohamed Abdelaal, Christian Hammacher, Harald Schöning (2023). REIN: A Comprehensive Benchmark Framework for Data Cleaning Methods in ML Pipelines. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 499–511. doi: 10.48786/edbt.2023.43.

Ziawasch Abedjan, Lukasz Golab, Felix Naumann, Thorsten Papenbrock (2018). Data Profiling. Synthesis Lectures on Data Management, doi: 10.1007/978-3-031-01865-7.

Yasser Altowim, Dmitri V. Kalashnikov, Sharad Mehrotra (2014). Progressive Approach to Relational Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 11, pp. 999–1010. doi: 10.14778/2732967.2732975.

Hotham Altwaijry, Dmitri V. Kalashnikov, Sharad Mehrotra (2013). Query-Driven Approach to Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 14, pp. 1846–1857. doi: 10.14778/2556549.2556567.

Hotham Altwaijry, Sharad Mehrotra, Dmitri V. Kalashnikov (2015). QuERy: A Framework for Integrating Entity Resolution with Query Processing. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 3, pp. 120–131. doi: 10.14778/2850583.2850587.

Alessia Amelio, Clara Pizzuti (2013). Average Common Submatrix: A New Image Distance Measure. *Proceedings of the International Conference on Image Analysis and Processing (ICIAP), Part 1.* Lecture Notes in Computer Science, vol. 8156, pp. 170–180. doi: 10.1007/978-3-642-41181-6_18.

Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, Matei Zaharia (2020). Delta Lake: High-Performance ACID Table Storage over

Cloud Object Stores. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 13, no. 12, pp. 3411–3424. DOI: 10.14778/3415478.3415560.

Nikolaus AUGSTEN, Michael H. BÖHLEN (2013). Similarity Joins in Relational Database Systems. Synthesis Lectures on Data Management, DOI: 10.1007/978-3-031-01851-0.

Santiago Andrés AZCOITIA, Nikolaos LAOUTARIS (2022). A Survey of Data Marketplaces and Their Business Models. *ACM SIGMOD Record*, vol. 51, no. 3, pp. 18–29. DOI: 10.1145/3572751.3572755.

Fabio AZZALINI, Cinzia CAPPIELLO, Chiara CRISCUOLO, Camilla SANCRICCA, Letizia TANCA (2023). Data Quality and Data Ethics: Towards a Trade-off Evaluation. *Proceedings of the VLDB Workshops.* CEUR Workshop Proceedings, vol. 3462. URL: https://ceur-ws.org/Vol-3462/QDB1.pdf.

Chiara BACHECHI, Laura PO, Federica ROLLO (2022). Big Data Analytics and Visualization in Traffic Monitoring. *Big Data Research*, vol. 27, art. 100292. DOI: 10.1016/j.bdr.2021.100292.

Gilbert BADARO, Mohammed SAEED, Paolo PAPOTTI (2023). Transformers for Tabular Data Representation: A Survey of Models and Applications. *Transactions of the Association for Computational Linguistics (TACL)*, vol. 11, pp. 227–249. DOI: 10.1162/tacl_a_00544.

Andrea BARALDI, Francesco DEL BUONO, Francesco GUERRA, Matteo PAGANELLI, Maurizio VINCINI (2023). An Intrinsically Interpretable Entity Matching System. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 645–657. DOI: 10.48786/edbt.2023.54.

Andrea BARALDI, Francesco DEL BUONO, Matteo PAGANELLI, Francesco GUERRA (2021). Using Landmarks for Explaining Entity Matching Models. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 451–456. DOI: 10.5441/002/edbt.2021.50.

Nils BARLAUG (2023a). LEMON: Explainable Entity Matching. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 35, no. 8, pp. 8171–8184. DOI: 10.1109/TKDE.2022.3200644.

— (2023b). ShallowBlocker: Improving Set Similarity Joins for Blocking. *arXiv*, art. 2312.15835. DOI: 10.48550/arXiv.2312.15835.

Nils BARLAUG, Jon Atle GULLA (2021). Neural Networks for Entity Matching: A Survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 3, art. 52. DOI: 10.1145/3442200.

Malte BARTH, Tibor BLEIDT, Martin BÜSSEMEYER, Fabian HESEDING, Niklas KÖHNECKE, Tobias BLEIFUSS, Leon BORNEMANN, Dmitri V. KALASHNIKOV, Felix NAUMANN, Divesh SRIVASTAVA (2023). Detecting Stale Data in Wikipedia Infoboxes. *Proceedings of the International*

Conference on Extending Database Technology (EDBT), pp. 450–456. DOI: `10.48786/edbt.2023.36`.

Carlo Batini, Cinzia Cappiello, Chiara Francalanci, Andrea Maurino (2009). Methodologies for Data Quality Assessment and Improvement. *ACM Computing Surveys*, vol. 41, no. 3, art. 16. DOI: `10.1145/1541880.1541883`.

Ashwin Belle, Raghuram Thiagarajan, S. M. Reza Soroushmehr, Fatemeh Navidi, Daniel A. Beard, Kayvan Najarian (2015). Big Data Analytics in Healthcare. *BioMed Research International*, vol. 2015, art. 370194. DOI: `10.1155/2015/370194`.

Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, Jennifer Widom (2009). Swoosh: a generic approach to entity resolution. *VLDB Journal*, vol. 18, no. 1, pp. 255–276. DOI: `10.1007/S00778-008-0098-x`.

Sonia Bergamaschi, Domenico Beneventano, Federica Mandreoli, Riccardo Martoglia, Francesco Guerra, Mirko Orsini, Laura Po, Maurizio Vincini, Giovanni Simonini, Song Zhu, Luca Gagliardelli, Luca Magnotta (2018). From Data Integration to Big Data Integration. *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*. Studies in Big Data, vol. 31, pp. 43–59. DOI: `10.1007/978-3-319-61893-7_3`.

Sonia Bergamaschi, Silvana Castano, Maurizio Vincini (1999). Semantic Integration of Semistructured and Structured Data Sources. *ACM SIGMOD Record*, vol. 28, no. 1, pp. 54–59. DOI: `10.1145/309844.309897`.

Laure Berti-Équille, Javier Borge-Holthoefer (2015). Veracity of Data. Synthesis Lectures on Data Management, DOI: `10.1007/978-3-031-01855-8`.

Chandra Sekhar Bhagavatula, Thanapon Noraset, Doug Downey (2015). TabEL: Entity Linking in Web Tables. *Proceedings of the International Semantic Web Conference (ISWC), Part 1*. Lecture Notes in Computer Science, vol. 9366, pp. 425–441. DOI: `10.1007/978-3-319-25007-6_25`.

Garrett Birkhoff (1940). Lattice Theory. Colloquium Publications, vol. 25. DOI: `10.1090/coll/025`.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, Sebastian Hellmann (2009). DBpedia: A crystallization point for the Web of Data. *Journal of Web Semantics*, vol. 7, no. 3, pp. 154–165. DOI: `10.1016/j.websem.2009.07.002`.

Tobias Bleifuss, Leon Bornemann, Theodore Johnson, Dmitri V. Kalashnikov, Felix Naumann, Divesh Srivastava (2018). Exploring Change: A New Dimension of Data Analytics. *Proceedings* *of the VLDB Endowment (PVLDB)*, vol. 12, no. 2, pp. 85–98. DOI: 10.14778/3282495.3282496.

Tobias Bleifuss, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, Divesh Srivastava (2021a). Structured Object Matching across Web Page Revisions. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 1284–1295. DOI: 10.1109/ICDE51399.2021.00115.

— (2021b). The Secret Life of Wikipedia Tables. *Proceedings of the Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores (SEA Data @ VLDB)*. CEUR Workshop Proceedings, vol. 2929, pp. 20–26. URL: https://ceur-ws.org/Vol-2929/paper4.pdf.

Jens Bleiholder, Felix Naumann (2008). Data Fusion. *ACM Computing Surveys*, vol. 41, no. 1, art. 1. DOI: 10.1145/1456650.1456651.

Burton H. Bloom (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, vol. 13, no. 7, pp. 422–426. DOI: 10.1145/362686.362692.

Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, Nikolaos Konstantinou (2020). Dataset Discovery in Data Lakes. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 709–720. DOI: 10.1109/ICDE48307.2020.00067.

Erik Brynjolfsson, Kristina McElheran (2016). The Rapid Adoption of Data-Driven Decision-Making. *American Economic Review*, vol. 106, no. 5, pp. 133–139. DOI: 10.1257/aer.p20161016.

Lukas Budach, Moritz Feuerpfeil, Nina Ihde, Andrea Nathansen, Nele Noack, Hendrik Patzlaff, Felix Naumann, Hazar Harmouch (2022). The Effects of Data Quality on Machine Learning Performance. *arXiv*, art. 2207.14529. DOI: 10.48550/arXiv.2207.14529.

Siarhei Bykau, Flip Korn, Divesh Srivastava, Yannis Velegrakis (2015). Fine-Grained Controversy Detection in Wikipedia. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 1573–1584. DOI: 10.1109/ICDE.2015.7113426.

Michael J. Cafarella, Alon Halevy, Nodira Khoussainova (2009). Data Integration for the Relational Web. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 1090–1101. DOI: 10.14778/1687627.1687750.

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang (2008). WebTables: Exploring the Power of Tables on the Web. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, pp. 538–549. DOI: 10.14778/1453856.1453916.

Riccardo Cappuzzo, Paolo Papotti, Saravanan Thirumuruganathan (2020). Creating Embeddings of Heterogeneous Relational Datasets for

Data Integration Tasks. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 1335–1349. DOI: 10.1145/3318464.3389742.

Raul CASTRO FERNANDEZ, Ziawasch ABEDJAN, Famien KOKO, Gina YUAN, Sam MADDEN, Michael STONEBRAKER (2018). Aurum: A Data Discovery System. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 1001–1012. DOI: 10.1109/ICDE.2018.00094.

Surajit CHAUDHURI, Umeshwar DAYAL (1997). An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74. DOI: 10.1145/248603.248616.

Peter CHRISTEN (2012a). A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 9, pp. 1537–1555. DOI: 10.1109/TKDE.2011.127.

— (2012b). Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Data-Centric Systems and Applications, DOI: 10.1007/978-3-642-31164-2.

Vassilis CHRISTOPHIDES, Vasilis EFTHYMIOU, Themis PALPANAS, George PAPADAKIS, Kostas STEFANIDIS (2021). An Overview of End-to-End Entity Resolution for Big Data. *ACM Computing Surveys*, vol. 53, no. 6, art. 127. DOI: 10.1145/3418896.

Xu CHU, Ihab F. ILYAS, Paraschos KOUTRIS (2016). Distributed Data Deduplication. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 11, pp. 864–875. DOI: 10.14778/2983200.2983203.

Jeffrey COHEN, Brian DOLAN, Mark DUNLAP, Joseph M. HELLERSTEIN, Caleb WELTON (2009). MAD Skills: New Analysis Practices for Big Data. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2, pp. 1481–1492. DOI: 10.14778/1687553.1687576.

Valter CRESCENZI, Andrea DE ANGELIS, Donatella FIRMANI, Maurizio MAZZEI, Paolo MERIALDO, Federico PIAI, Divesh SRIVASTAVA (2021). Alaska: A Flexible Benchmark for Data Integration Tasks. *arXiv*, art. 2101.11259. DOI: 10.48550/arXiv.2101.11259.

Kenneth CUKIER, Viktor MAYER-SCHOENBERGER (2013). The Rise of Big Data: How It's Changing the Way We Think About the World. *Foreign Affairs*, vol. 92, no. 3, pp. 28–40. URL: https://www.foreignaffairs.com/articles/2013-04-03/rise-big-data.

Sanjib DAS, Paul SUGANTHAN G. C., AnHai DOAN, Jeffrey F. NAUGHTON, Ganesh KRISHNAN, Rohit DEEP, Esteban ARCAUTE, Vijay RAGHAVENDRA, Youngchoon PARK (2017). Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. *Proceedings of the ACM*

International Conference on Management of Data (SIGMOD), pp. 1431–1446. DOI: 10.1145/3035918.3035960.

Anish Das Sarma, Xin Dong, Alon Halevy (2008). Bootstrapping Pay-As-You-Go Data Integration Systems. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 861–874. DOI: 10.1145/1376616.1376702.

Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, Cong Yu (2012). Finding Related Tables. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 817–828. DOI: 10.1145/2213836.2213962.

Fabien De Marchi, Stéphane Lopes, Jean-Marc Petit (2002). Efficient Algorithms for Mining Inclusion Dependencies. *Proceedings of the International Conference on Extending Database Technology (EDBT)*. Lecture Notes in Computer Science, vol. 2287, pp. 464–476. DOI: 10.1007/3-540-45876-X_30.

— (2009). Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, vol. 32, no. 1, pp. 53–73. DOI: 10.1007/s10844-007-0048-x.

Fabien De Marchi, Jean-Marc Petit (2003). Zigzag: a new algorithm for mining large inclusion dependencies in databases. *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pp. 27–34. DOI: 10.1109/ICDM.2003.1250899.

Dong Deng, Wenbo Tao, Ziawasch Abedjan, Ahmed Elmagarmid, Ihab F. Ilyas, Guoliang Li, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, Nan Tang (2019). Unsupervised String Transformation Learning for Entity Consolidation. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 196–207. DOI: 10.1109/ICDE.2019.00026.

Dong Deng, Yufei Tao, Guoliang Li (2018). Overlap Set Similarity Joins with Theoretical Guarantees. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 905–920. DOI: 10.1145/3183713.3183748.

Li Deng, Shuo Zhang, Krisztian Balog (2019). Table2Vec: Neural Word and Entity Embeddings for Table Population and Retrieval. *Proceedings of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 1029–1032. DOI: 10.1145/3331184.3331333.

Xiang Deng, Huan Sun, Alyssa Lees, You Wu, Cong Yu (2020). TURL: Table Understanding through Representation Learning. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 3, pp. 307–319. DOI: 10.14778/3430915.3430921.

Jacob DEVLIN, Ming-Wei CHANG, Kenton LEE, Kristina TOUTANOVA (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), Volume 1 (Long and Short Papers)*, pp. 4171–4186. DOI: `10.18653/v1/n19-1423`.

Vincenzo DI CICCO, Donatella FIRMANI, Nick KOUDAS, Paolo MERIALDO, Divesh SRIVASTAVA (2019). Interpreting Deep Learning Models for Entity Resolution: An Experience Report Using LIME. *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM @ SIGMOD)*, art. 8. DOI: `10.1145/3329859.3329878`.

AnHai DOAN, Pradap KONDA, Paul SUGANTHAN G. C., Yash GOVIND, Derek PAULSEN, Kaushik CHANDRASEKHAR, Philip MARTINKUS, Matthew CHRISTIE (2020). Magellan: Toward Building Ecosystems of Entity Matching Solutions. *Communications of the ACM*, vol. 63, no. 8, pp. 83–91. DOI: `10.1145/3405476`.

Dany DOIRON, Paul BURTON, Yannick MARCON, Amadou GAYE, Bruce H. R. WOLFFENBUTTEL, Markus PEROLA, Ronald P. STOLK, Luisa FOCO, Cosetta MINELLI, Melanie WALDENBERGER, Rolf HOLLE, Kirsti KVALØY, Hans L. HILLEGE, Anne-Marie TASSÉ, Vincent FERRETTI, Isabel FORTIER (2013). Data harmonization and federated analysis of population-based studies: the BioSHaRE project. *Emerging Themes in Epidemiology*, vol. 10, art. 12. DOI: `10.1186/1742-7622-10-12`.

Xin Luna DONG, Divesh SRIVASTAVA (2015). Big Data Integration. Synthesis Lectures on Data Management, DOI: `10.1007/978-3-031-01853-4`.

Yuyang DONG, Kunihiro TAKEOKA, Chuan XIAO, Masafumi OYAMADA (2021). Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 456–467. DOI: `10.1109/ICDE51399.2021.00046`.

Yuyang DONG, Chuan XIAO, Takuma NOZAWA, Masafumi ENOMOTO, Masafumi OYAMADA (2023). DeepJoin: Joinable Table Discovery with Pre-trained Language Models. *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2458–2470. DOI: `10.14778/3603581.3603587`.

Halbert L. DUNN (1946). Record Linkage. *American Journal of Public Health*, vol. 36, no. 12, pp. 1412–1416. DOI: `10.2105/AJPH.36.12.1412`.

Falco DÜRSCH, Axel STEBNER, Fabian WINDHEUSER, Maxi FISCHER, Tim FRIEDRICH, Nils STRELOW, Tobias BLEIFUSS, Hazar HARMOUCH, Lan JIANG, Thorsten PAPENBROCK, Felix NAUMANN (2019). Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen

Algorithms. *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 219–228. DOI: 10.1145/3357384.3357916.

Julian EBERIUS, Katrin BRAUNSCHWEIG, Markus HENTSCH, Maik THIELE, Ahmad AHMADOV, Wolfgang LEHNER (2015). Building the Dresden Web Table Corpus: A Classification Approach. *Proceedings of the IEEE/ACM International Symposium on Big Data Computing (BDC)*, pp. 41–50. DOI: 10.1109/BDC.2015.30.

Muhammad EBRAHEEM, Saravanan THIRUMURUGANATHAN, Shafiq JOTY, Mourad OUZZANI, Nan TANG (2018). Distributed Representations of Tuples for Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, pp. 1454–1467. DOI: 10.14778/3236187.3236198.

Vasilis EFTHYMIOU, Kostas STEFANIDIS, Evaggelia PITOURA, Vassilis CHRISTOPHIDES (2021). FairER: Entity Resolution With Fairness Constraints. *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 3004–3008. DOI: 10.1145/3459637.3482105.

Lisa EHRLINGER, Wolfram WÖSS (2022). A Survey of Data Quality Measurement and Monitoring Tools. *Frontiers in Big Data*, vol. 5, art. 850611. DOI: 10.3389/fdata.2022.850611.

Ahmed K. ELMAGARMID, Panagiotis G. IPEIROTIS, Vassilios S. VERYKIOS (2007). Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, no. 1, pp. 1–16. DOI: 10.1109/TKDE.2007.250581.

Mahdi ESMAILOGHLI, Jorge-Arnulfo QUIANÉ-RUIZ, Ziawasch ABEDJAN (2022). MATE: Multi-Attribute Table Extraction. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 8, pp. 1684–1696. DOI: 10.14778/3529337.3529353.

Ju FAN, Meiyu LU, Beng Chin OOI, Wang-Chiew TAN, Meihui ZHANG (2014). A Hybrid Machine-Crowdsourcing System for Matching Web Tables. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 976–987. DOI: 10.1109/ICDE.2014.6816716.

Wenfei FAN, Floris GEERTS (2012). Foundations of Data Quality Management. Synthesis Lectures on Data Management, DOI: 10.1007/978-3-031-01892-3.

Ivan P. FELLEGI, Alan B. SUNTER (1969). A Theory for Record Linkage. *Journal of the American Statistical Association*, vol. 64, no. 328, pp. 1183–1210. DOI: 10.1080/01621459.1969.10501049.

Alvaro A. A. FERNANDES, Martin KOEHLER, Nikolaos KONSTANTINOU, Pavel PANKIN, Norman W. PATON, Rizos SAKELLARIOU (2023). Data

Preparation: A Technological Perspective and Review. *SN Computer Science*, vol. 4, no. 4, art. 425. DOI: 10.1007/s42979-023-01828-8.

Donatella FIRMANI, Barna SAHA, Divesh SRIVASTAVA (2016). Online Entity Resolution Using an Oracle. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 5, pp. 384–395. DOI: 10.14778/2876473.2876474.

Tim FURCHE, Georg GOTTLOB, Leonid LIBKIN, Giorgio ORSI, Norman W. PATON (2016). Data Wrangling for Big Data: Challenges and Opportunities. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 473–478. DOI: 10.5441/002/edbt.2016.44.

Luca GAGLIARDELLI, George PAPADAKIS, Giovanni SIMONINI, Sonia BERGAMASCHI, Themis PALPANAS (2022a). Generalized Supervised Meta-blocking. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 9, pp. 1902–1910. DOI: 10.14778/3538598.3538611.

Luca GAGLIARDELLI, Giovanni SIMONINI, Domenico BENEVENTANO, Sonia BERGAMASCHI (2019). SparkER: Scaling Entity Resolution in Spark. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 602–605. DOI: 10.5441/002/edbt.2019.66.

Luca GAGLIARDELLI, Giovanni SIMONINI, Sonia BERGAMASCHI (2020). RulER: Scaling Up Record-level Matching Rules. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pp. 611–614. DOI: 10.5441/002/edbt.2020.76.

Luca GAGLIARDELLI, Luca ZECCHINI, Domenico BENEVENTANO, Giovanni SIMONINI, Sonia BERGAMASCHI, Mirko ORSINI, Luca MAGNOTTA, Emma MESCOLI, Andrea LIVALDI, Nicola GESSA, Piero DE SABBATA, Gianluca D'AGOSTA, Fabrizio PAOLUCCI, Fabio MORETTI (2022b). ECDP: A Big Data Platform for the Smart Monitoring of Local Energy Communities. *Proceedings of the EDBT/ICDT Workshops.* CEUR Workshop Proceedings, vol. 3135. URL: http://ceur-ws.org/Vol-3135/dataplat_short4.pdf.

Luca GAGLIARDELLI, Luca ZECCHINI, Luca FERRETTI, Domenico BENEVENTANO, Giovanni SIMONINI, Sonia BERGAMASCHI, Mirko ORSINI, Luca MAGNOTTA, Emma MESCOLI, Andrea LIVALDI, Nicola GESSA, Piero DE SABBATA, Gianluca D'AGOSTA, Fabrizio PAOLUCCI, Fabio MORETTI (2023). A big data platform exploiting auditable tokenization to promote good practices inside local energy communities. *Future Generation Computer Systems (FGCS)*, vol. 141, pp. 595–610. DOI: 10.1016/j.future.2022.12.007.

David GALE, Lloyd S. SHAPLEY (1962). College Admissions and the Stability of Marriage. *American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15. DOI: 10.2307/2312726.

Sainyam Galhotra, Donatella Firmani, Barna Saha, Divesh Srivastava (2021). Efficient and effective ER with progressive blocking. *VLDB Journal*, vol. 30, no. 4, pp. 537–557. doi: 10.1007/S00778-021-00656-7.

Michael R. Garey, David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness.

Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude Shavlik, Xiaojin Zhu (2014). Corleone: Hands-Off Crowdsourcing for Entity Matching. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 601–612. doi: 10.1145/2588555.2588576.

Zhenzhen Gu, Francesco Corcoglioniti, Davide Lanti, Alessandro Mosca, Guohui Xiao, Jing Xiong, Diego Calvanese (2022). A systematic overview of data federation systems. *Semantic Web*, vol. 15, no. 1, pp. 107–165. doi: 10.3233/SW-223201.

Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, Dino Pedreschi (2019). A Survey of Methods for Explaining Black Box Models. *ACM Computing Surveys*, vol. 51, no. 5, art. 93. doi: 10.1145/3236009.

Mazhar Hameed, Felix Naumann (2020). Data Preparation: A Survey of Commercial Tools. *ACM SIGMOD Record*, vol. 49, no. 3, pp. 18–29. doi: 10.1145/3444831.3444835.

Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, Renée J. Miller (2009). Framework for Evaluating Clustering Algorithms in Duplicate Detection. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 1282–1293. doi: 10.14778/1687627.1687771.

Anders Haug, Frederik Zachariassen, Dennis van Liempd (2011). The costs of poor data quality. *Journal of Industrial Engineering and Management*, vol. 4, no. 2, pp. 168–193. doi: 10.3926/jiem.2011.v4n2.p168-193.

Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, Julian Martin Eisenschlos (2020). TaPas: Weakly Supervised Table Parsing via Pre-training. *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4320–4333. doi: 10.18653/v1/2020.acl-main.398.

Xuedong Huang, James Baker, Raj Reddy (2014). A Historical Perspective of Speech Recognition. *Communications of the ACM*, vol. 57, no. 1, pp. 94–103. doi: 10.1145/2500887.

Madelon Hulsebos, Çağatay Demiralp, Paul Groth (2023). GitTables: A Large-Scale Corpus of Relational Tables. *Proceedings of the ACM on Management of Data (PACMMOD)*, vol. 1, no. 1, art. 30. doi: 10.1145/3588710.

Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çağatay Demiralp, César Hidalgo (2019). Sherlock: A Deep Learning Approach to Semantic Data Type Detection. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1500–1508. doi: 10.1145/3292500.3330993.

Hiroshi Iida, Dung Thai, Varun Manjunatha, Mohit Iyyer (2021). TABBIE: Pretrained Representations of Tabular Data. *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 3446–3456. doi: 10.18653/v1/2021.naacl-main.270.

Ihab F. Ilyas, Xu Chu (2019). Data Cleaning. doi: 10.1145/3310205.

William H. Inmon (1992). Building the Data Warehouse.

Paul Jaccard (1912). The Distribution of the Flora in the Alpine Zone. *New Phytologist*, vol. 11, no. 2, pp. 37–50. doi: 10.1111/j.1469-8137.1912.tb05611.x.

Mohammad Hossein Jarrahi, Ali Memariani, Shion Guha (2023). The Principles of Data-Centric AI. *Communications of the ACM*, vol. 66, no. 8, pp. 84–92. doi: 10.1145/3571724.

Youri Kaminsky, Eduardo H. M. Pena, Felix Naumann (2023). Discovering Similarity Inclusion Dependencies. *Proceedings of the ACM on Management of Data (PACMMOD)*, vol. 1, no. 1, art. 75. doi: 10.1145/3588929.

Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, Renée J. Miller (2022). Integrating Data Lake Tables. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 4, pp. 932–945. doi: 10.14778/3574245.3574274.

Gang-Hoon Kim, Silvana Trimi, Ji-Hyong Chung (2014). Big-Data Applications in the Government Sector. *Communications of the ACM*, vol. 57, no. 3, pp. 78–85. doi: 10.1145/2500873.

Maximilian Koch, Mahdi Esmailoghli, Sören Auer, Ziawasch Abedjan (2023). Duplicate Table Detection with Xash. *Proceedings of the Conference on Database Systems for Business, Technology and Web (BTW)*. Lecture Notes in Informatics, vol. P-331, pp. 367–390. doi: 10.18420/BTW2023-18.

Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, Ganesh Krishnan, Rohit Deep, Vijay Raghavendra (2016). Magellan: Toward Building Entity Matching Management Systems. *Proceedings of the*

VLDB Endowment (PVLDB), vol. 9, no. 12, pp. 1197–1208. DOI: 10.14778/2994509.2994535.

Lukas LASKOWSKI, Florian SOLD (2023). Explainable Data Matching: Selecting Representative Pairs with Active Learning Pair-Selection Strategies. *Proceedings of the Conference on Database Systems for Business, Technology and Web (BTW)*. Lecture Notes in Informatics, vol. P-331, pp. 1099–1104. DOI: 10.18420/BTW2023-77.

Yann LECUN, Yoshua BENGIO, Geoffrey HINTON (2015). Deep learning. *Nature*, vol. 521, pp. 436–444. DOI: 10.1038/nature14539.

Oliver LEHMBERG, Christian BIZER (2017). Stitching Web Tables for Improving Matching Quality. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, no. 11, pp. 1502–1513. DOI: 10.14778/3137628.3137657.

Oliver LEHMBERG, Dominique RITZE, Robert MEUSEL, Christian BIZER (2016). A Large Public Corpus of Web Tables containing Time and Context Metadata. *Proceedings of the International World Wide Web Conference (WWW), Companion Volume*, pp. 75–76. DOI: 10.1145/2872518.2889386.

Jure LESKOVEC, Anand RAJARAMAN, Jeffrey David ULLMAN (2020). Mining of Massive Datasets. URL: http://www.mmds.org.

Guoliang LI (2017). Human-in-the-loop Data Integration. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 10, no. 12, pp. 2006–2017. DOI: 10.14778/3137765.3137833.

Peng LI, Xi RAO, Jennifer BLASE, Yue ZHANG, Xu CHU, Ce ZHANG (2021). CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 13–24. DOI: 10.1109/ICDE51399.2021.00009.

Xian LI, Xin Luna DONG, Kenneth LYONS, Weiyi MENG, Divesh SRIVASTAVA (2012). Truth Finding on the Deep Web: Is the Problem Solved? *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 2, pp. 97–108. DOI: 10.14778/2535568.2448943.

Yuliang LI, Jinfeng LI, Yoshihiko SUHARA, AnHai DOAN, Wang-Chiew TAN (2020). Deep Entity Matching with Pre-Trained Language Models. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 1, pp. 50–60. DOI: 10.14778/3421424.3421431.

Girija LIMAYE, Sunita SARAWAGI, Soumen CHAKRABARTI (2010). Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 1338–1347. DOI: 10.14778/1920841.1921005.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv*, art. 1907.11692. doi: `10.48550/arXiv.1907.11692`.

Stéphane Lopes, Jean-Marc Petit, Farouk Toumani (2002). Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, vol. 27, no. 1, pp. 1–19. doi: `10.1016/S0306-4379(01)00027-8`.

David Loshin (2008). Master Data Management. doi: `10.1016/B978-0-12-374225-4.X0001-X`.

Michael Loster, Ioannis Koumarelas, Felix Naumann (2021). Knowledge Transfer for Entity Resolution with Siamese Neural Networks. *ACM Journal of Data and Information Quality (JDIQ)*, vol. 13, no. 1, art. 2. doi: `10.1145/3410157`.

Bruce T. Lowerre (1976). The HARPY Speech Recognition System. PhD thesis. Carnegie Mellon University.

Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin Luna Dong, David Ko, Cong Yu, Alon Halevy (2007). Web-scale Data Integration: You can afford to Pay As You Go. *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 342–350. url: `http://cidrdb.org/cidr2007/papers/cidr07p40.pdf`.

Mohsen Marjani, Fariza Nasaruddin, Abdullah Gani, Ahmad Karim, Ibrahim Abaker Targio Hashem, Aisha Siddiqa, Ibrar Yaqoob (2017). Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, vol. 5, pp. 5247–5261. doi: `10.1109/ACCESS.2017.2689040`.

Andrew McCallum, Kamal Nigam, Lyle H. Ungar (2000). Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 169–178. doi: `10.1145/347090.347123`.

Vamsi Meduri, Lucian Popa, Prithviraj Sen, Mohamed Sarwat (2020). A Comprehensive Benchmark Framework for Active Learning Methods in Entity Matching. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 1133–1147. doi: `10.1145/3318464.3380597`.

Renée J. Miller (2018). Open Data Integration. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 12, pp. 2130–2139. doi: `10.14778/3229863.3240491`.

H. Mirzaalian, L. Ning, P. Savadjiev, O. Pasternak, S. Bouix, O. Michailovich, G. Grant, C. E. Marx, R. A. Morey, L. A. Flash-

MAN, M. S. GEORGE, T. W. MCALLISTER, N. ANDALUZ, L. SHUTTER, R. COIMBRA, R. D. ZAFONTE, M. J. COLEMAN, M. KUBICKI, C. F. WESTIN, M. B. STEIN, M. E. SHENTON, Y. RATHI (2016). Inter-site and inter-scanner diffusion MRI data harmonization. *NeuroImage*, vol. 135, pp. 311–323. DOI: `10.1016/j.neuroimage.2016.04.041`.

Sidharth MUDGAL, Han LI, Theodoros REKATSINAS, AnHai DOAN, Youngchoon PARK, Ganesh KRISHNAN, Rohit DEEP, Esteban ARCAUTE, Vijay RAGHAVENDRA (2018). Deep Learning for Entity Matching: A Design Space Exploration. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 19–34. DOI: `10.1145/3183713.3196926`.

Avanika NARAYAN, Ines CHAMI, Laurel ORR, Christopher RÉ (2022). Can Foundation Models Wrangle Your Data? *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 4, pp. 738–746. DOI: `10.14778/3574245.3574258`.

Fatemeh NARGESIAN, Ken Q. PU, Erkang ZHU, Bahar GHADIRI BASHARDOOST, Renée J. MILLER (2020). Organizing Data Lakes for Navigation. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 1939–1950. DOI: `10.1145/3318464.3380605`.

Fatemeh NARGESIAN, Erkang ZHU, Renée J. MILLER, Ken Q. PU, Patricia C. AROCENA (2019). Data Lake Management: Challenges and Opportunities. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 12, no. 12, pp. 1986–1989. DOI: `10.14778/3352063.3352116`.

Fatemeh NARGESIAN, Erkang ZHU, Ken Q. PU, Renée J. MILLER (2018). Table Union Search on Open Data. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 7, pp. 813–825. DOI: `10.14778/3192965.3192973`.

Felix NAUMANN, Melanie HERSCHEL (2010). An Introduction to Duplicate Detection. Synthesis Lectures on Data Management, DOI: `10.1007/978-3-031-01835-0`.

Konstantinos NIKOLETOS, George PAPADAKIS, Manolis KOUBARAKIS (2022). pyJedAI: a Lightsaber for Link Discovery. *Proceedings of the ISWC Posters, Demos and Industry Tracks.* CEUR Workshop Proceedings, vol. 3254. URL: `https://ceur-ws.org/Vol-3254/paper366.pdf`.

Kevin O'HARE, Anna JUREK-LOUGHREY, Cassio DE CAMPOS (2019). A Review of Unsupervised and Semi-supervised Blocking Methods for Record Linkage. *Linking and Mining Heterogeneous and Multi-view Data.* Unsupervised and Semi-Supervised Learning, pp. 79–105. DOI: `10.1007/978-3-030-01872-6_4`.

Matteo PAGANELLI, Francesco DEL BUONO, Andrea BARALDI, Francesco GUERRA (2022). Analyzing How BERT Performs Entity Matching. *Pro-*

ceedings of the VLDB Endowment (PVLDB), vol. 15, no. 8, pp. 1726–1738. DOI: 10.14778/3529337.3529356.

George PAPADAKIS, Ekaterini IOANNOU, Claudia NIEDERÉE, Peter FANKHAUSER (2011). Efficient Entity Resolution for Large Heterogeneous Information Spaces. *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM)*, pp. 535–544. DOI: 10.1145/1935826.1935903.

George PAPADAKIS, Ekaterini IOANNOU, Emanouil THANOS, Themis PALPANAS (2021a). The Four Generations of Entity Resolution. Synthesis Lectures on Data Management, DOI: 10.1007/978-3-031-01878-7.

George PAPADAKIS, Georgia KOUTRIKA, Themis PALPANAS, Wolfgang NEJDL (2014a). Meta-Blocking: Taking Entity Resolution to the Next Level. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 26, no. 8, pp. 1946–1960. DOI: 10.1109/TKDE.2013.54.

George PAPADAKIS, George MANDILARAS, Luca GAGLIARDELLI, Giovanni SIMONINI, Emmanouil THANOS, George GIANNAKOPOULOS, Sonia BERGAMASCHI, Themis PALPANAS, Manolis KOUBARAKIS (2020). Three-dimensional Entity Resolution with JedAI. *Information Systems*, vol. 93, art. 101565. DOI: 10.1016/j.is.2020.101565.

George PAPADAKIS, George PAPASTEFANATOS, Georgia KOUTRIKA (2014b). Supervised Meta-blocking. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 14, pp. 1929–1940. DOI: 10.14778/2733085.2733098.

George PAPADAKIS, Dimitrios SKOUTAS, Emmanouil THANOS, Themis PALPANAS (2021b). Blocking and Filtering Techniques for Entity Resolution: A Survey. *ACM Computing Surveys*, vol. 53, no. 2, art. 31. DOI: 10.1145/3377455.

George PAPADAKIS, Jonathan SVIRSKY, Avigdor GAL, Themis PALPANAS (2016). Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 9, pp. 684–695. DOI: 10.14778/2947618.2947624.

George PAPADAKIS, Leonidas TSEKOURAS, Emmanouil THANOS, George GIANNAKOPOULOS, Themis PALPANAS, Manolis KOUBARAKIS (2019). Domain- and Structure-Agnostic End-to-End Entity Resolution with JedAI. *ACM SIGMOD Record*, vol. 48, no. 4, pp. 30–36. DOI: 10.1145/3385658.3385664.

Thorsten PAPENBROCK, Arvid HEISE, Felix NAUMANN (2015a). Progressive Duplicate Detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 5, pp. 1316–1329. DOI: 10.1109/TKDE.2014.2359666.

Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, Felix Naumann (2015b). Divide & Conquer-based Inclusion Dependency Discovery. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 7, pp. 774–785. doi: 10.14778/2752939.2752946.

Norman W. Paton, Jiaoyan Chen, Zhenyu Wu (2023). Dataset Discovery and Exploration: A Survey. *ACM Computing Surveys*, vol. 56, no. 4, art. 102. doi: 10.1145/3626521.

Derek Paulsen, Yash Govind, AnHai Doan (2023). Sparkly: A Simple yet Surprisingly Strong TF/IDF Blocker for Entity Matching. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 6, pp. 1507–1519. doi: 10.14778/3583140.3583163.

Ralph Peeters, Christian Bizer (2023). Using ChatGPT for Entity Matching. *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS): Short Papers, Doctoral Consortium and Workshops*. Communications in Computer and Information Science, vol. 1850, pp. 221–230. doi: 10.1007/978-3-031-42941-5_20.

Thomas Pellissier Tanon, Gerhard Weikum, Fabian Suchanek (2020). YAGO 4: A Reason-able Knowledge Base. *Proceedings of the Extended Semantic Web Conference (ESWC)*. Lecture Notes in Computer Science, vol. 12123, pp. 583–596. doi: 10.1007/978-3-030-49461-2_34.

Alberto Pietrangelo, Giovanni Simonini, Sonia Bergamaschi, Ioannis Koumarelas, Felix Naumann (2018). Towards Progressive Search-driven Entity Resolution. *Proceedings of the Italian Symposium on Advanced Database Systems (2018)*. CEUR Workshop Proceedings, vol. 2161. url: https://ceur-ws.org/Vol-2161/paper16.pdf.

Aleš Popovič, Ray Hackney, Rana Tassabehji, Mauro Castelli (2018). The impact of big data analytics on firms' high value business performance. *Information Systems Frontiers*, vol. 20, no. 2, pp. 209–222. doi: 10.1007/s10796-016-9720-4.

Martin Potthast, Benno Stein, Robert Gerling (2008). Automatic Vandalism Detection in Wikipedia. *Proceedings of the European Conference on Information Retrieval (ECIR)*. Lecture Notes in Computer Science, vol. 4956, pp. 663–668. doi: 10.1007/978-3-540-78646-7_75.

Gil Press (2016). Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says. *Forbes* (Mar. 23, 2016). url: https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/.

Erhard Rahm, Philip A. Bernstein (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, vol. 10, no. 4, pp. 334–350. doi: 10.1007/s007780100057.

Marco Túlio Ribeiro, Sameer Singh, Carlos Guestrin (2016). ”Why Should I Trust You?”: Explaining the Predictions of Any Classifier. *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 1135–1144. doi: `10.1145/2939672.2939778`.

Dominique Ritze, Oliver Lehmberg, Christian Bizer (2015). Matching HTML Tables to DBpedia. *Proceedings of the ACM International Conference on Web Intelligence, Mining and Semantics (WIMS)*, art. 10. doi: `10.1145/2797115.2797118`.

Betsy Rolland, Suzanna Reid, Deanna Stelling, Greg Warnick, Mark Thornquist, Ziding Feng, John D. Potter (2015). Toward Rigorous Data Harmonization in Cancer Epidemiology Research: One Approach. *American Journal of Epidemiology*, vol. 182, no. 12, pp. 1033–1038. doi: `10.1093/aje/kwv133`.

Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, Lora Aroyo (2021). ”Everyone wants to do the model work, not the data work”: Data Cascades in High-Stakes AI. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, art. 39. doi: `10.1145/3411764.3445518`.

Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv*, art. 1910.01108. doi: `10.48550/arXiv.1910.01108`.

Enrico Sartori, Yannis Velegrakis, Francesco Guerra (2016). Entity-Based Keyword Search in Web Documents. *Transactions on Computational Collective Intelligence*, vol. 21, pp. 21–49. doi: `10.1007/978-3-662-49521-6_2`.

Burr Settles (2012). Active Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning, doi: `10.1007/978-3-031-01560-1`.

Nima Shahbazi, Nikola Danevski, Fatemeh Nargesian, Abolfazl Asudeh, Divesh Srivastava (2023). Through the Fairness Lens: Experimental Analysis and Evaluation of Entity Matching. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 11, pp. 3279–3292. doi: `10.14778/3611479.3611525`.

Giovanni Simonini, Sonia Bergamaschi, H. V. Jagadish (2016). BLAST: a Loosely Schema-aware Meta-blocking Approach for Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 12, pp. 1173–1184. doi: `10.14778/2994509.2994533`.

Giovanni Simonini, Luca Gagliardelli, Sonia Bergamaschi, H. V. Jagadish (2019). Scaling entity resolution: A loosely schema-aware approach. *Information Systems*, vol. 83, pp. 145–165. doi: `10.1016/j.is.2019.03.006`.

Giovanni Simonini, George Papadakis, Themis Palpanas, Sonia Bergamaschi (2018). Schema-agnostic Progressive Entity Resolution. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 53–64. doi: `10.1109/ICDE.2018.00015`.

Giovanni Simonini, Luca Zecchini, Sonia Bergamaschi, Felix Naumann (2022). Entity Resolution On-Demand. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 15, no. 7, pp. 1506–1518. doi: `10.14778/3523210.3523226`.

Giovanni Simonini, Luca Zecchini, Felix Naumann, Sonia Bergamaschi (2023). Entity Resolution On-Demand for Querying Dirty Datasets. *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD)*. CEUR Workshop Proceedings, vol. 3478, pp. 410–419. url: `https://ceur-ws.org/Vol-3478/paper70.pdf`.

Khanin Sisaengsuwanchai, Navapat Nananukul, Mayank Kejriwal (2023). How does prompt engineering affect ChatGPT performance on unsupervised entity resolution? *arXiv*, art. 2310.06174. doi: `10.48550/arXiv.2310.06174`.

Paolo Sottovia, Matteo Paganelli, Francesco Guerra, Yannis Velegrakis (2019). Finding Synonymous Attributes in Evolving Wikipedia Infoboxes. *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*. Lecture Notes in Computer Science, vol. 11695, pp. 169–185. doi: `10.1007/978-3-030-28730-6_11`.

Kavitha Srinivas, Julian Dolby, Ibrahim Abdelaziz, Oktie Hassanzadeh, Harsha Kokel, Aamod Khatiwada, Tejaswini Pedapati, Subhajit Chaudhury, Horst Samulowitz (2023). LakeBench: Benchmarks for Data Discovery over Data Lakes. *arXiv*, art. 2307.04217. doi: `10.48550/arXiv.2307.04217`.

Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stan Zdonik, Alexander Pagan, Shan Xu (2013). Data Curation at Scale: The Data Tamer System. *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. url: `http://cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf`.

Nesime Tatbul (2010). Streaming Data Integration: Challenges and Opportunities. *Proceedings of the ICDE Workshops*, pp. 155–158. doi: `10.1109/ICDEW.2010.5452751`.

Tommaso Teofili, Donatella Firmani, Nick Koudas, Vincenzo Martello, Paolo Merialdo, Divesh Srivastava (2022). Effective Explanations for Entity Resolution Models. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 2709–2721. doi: `10.1109/ICDE53745.2022.00248`.

Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, AnHai Doan (2021). Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 14, no. 11, pp. 2459–2472. doi: 10.14778/3476249.3476294.

Elisabeth R. M. Tillier, Robert L. Charlebois (2009). The human protein coevolution network. *Genome Research*, vol. 19, no. 10, pp. 1861–1871. doi: 10.1101/gr.092452.109.

Fabian Tschirschnitz, Thorsten Papenbrock, Felix Naumann (2017). Detecting Inclusion Dependencies on Very Many Tables. *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, art. 18. doi: 10.1145/3105959.

Panos Vassiliadis, Alkis Simitsis, Spiros Skiadopoulos (2002). Conceptual Modeling for ETL Processes. *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)*, pp. 14–21. doi: 10.1145/583890.583893.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin (2017). Attention Is All You Need. *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*. Advances in Neural Information Processing Systems, vol. 30, pp. 5998–6008. url: http://papers.nips.cc/paper/7181-attention-is-all-you-need.

Liane Vogel, Carsten Binnig (2023). WikiDBs: A Corpus Of Relational Databases From Wikidata. *Proceedings of the VLDB Workshops*. CEUR Workshop Proceedings, vol. 3462. url: https://ceur-ws.org/Vol-3462/TADA3.pdf.

Jin Wang, Yuliang Li (2022). Minun: Evaluating Counterfactual Explanations for Entity Matching. *Proceedings of the Workshop on Data Management for End-To-End Machine Learning (DEEM @ SIGMOD)*, art. 7. doi: 10.1145/3533028.3533304.

Steven Euijong Whang, Hector Garcia-Molina (2012). Joint Entity Resolution. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 294–305. doi: 10.1109/ICDE.2012.119.

Steven Euijong Whang, David Marmaros, Hector Garcia-Molina (2013). Pay-As-You-Go Entity Resolution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 5, pp. 1111–1124. doi: 10.1109/TKDE.2012.43.

Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, Saravanan Thirumuruganathan (2020). ZeroER: Entity Resolution using Zero Labeled Examples. *Proceedings of the ACM International Conference on Man-*

agement of Data (SIGMOD), pp. 1149–1164. DOI: `10.1145/3318464.3389743`.

Pengcheng Yin, Graham Neubig, Wen-tau Yih, Sebastian Riedel (2020). TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 8413–8426. DOI: `10.18653/v1/2020.acl-main.745`.

Luca Zecchini, Tobias Bleifuss, Giovanni Simonini, Sonia Bergamaschi, Felix Naumann (2024). Determining the Largest Overlap between Tables. *Proceedings of the ACM on Management of Data (PACMMOD)*, vol. 2, no. 1, art. 48. DOI: `10.1145/3639303`.

Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi (2020). Entity Resolution on Camera Records without Machine Learning. *Proceedings of the International Workshop on Challenges and Experiences from Data Integration to Knowledge Graphs (DI2KG @ VLDB)*. CEUR Workshop Proceedings, vol. 2726. URL: `https://ceur-ws.org/Vol-2726/paper3.pdf`.

Luca Zecchini, Giovanni Simonini, Sonia Bergamaschi, Felix Naumann (2023). BrewER: Entity Resolution On-Demand. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 16, no. 12, pp. 4026–4029. DOI: `10.14778/3611540.3611612`.

Yi Zhang, Zachary G. Ives (2020). Finding Related Tables in Data Lakes for Interactive Data Science. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 1951–1966. DOI: `10.1145/3318464.3389726`.

Erkang Zhu, Dong Deng, Fatemeh Nargesian, Renée J. Miller (2019). JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pp. 847–864. DOI: `10.1145/3299869.3300065`.

Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, Renée J. Miller (2016). LSH Ensemble: Internet-Scale Domain Search. *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 12, pp. 1185–1196. DOI: `10.14778/2994509.2994534`.

Liang Zhu, Xu Du, Qin Ma, Weiyi Meng, Haibo Liu (2018). Keyword Search with Real-time Entity Resolution in Relational Databases. *Proceedings of the International Conference on Machine Learning and Computing (ICMLC)*, pp. 134–139. DOI: `10.1145/3195106.3195171`.