Università degli Studi di Modena e Reggio Emilia
FACOLTÀ DI INGEGNERIA "ENZO FERRARI"

Tesi di Laurea

A Query-Driven Approach to Entity Resolution based on Data Ordering

Giacomo Amici

Relatori:
Prof. Dr. Sonia Bergamaschi
Prof. Dr. Felix Naumann

Anno Accademico 2017 / 2018

Abstract

This thesis is the outcome of a research developed in collaboration with the Hasso-Plattner Institut of Postdam, Germany. In particular, I spent two months as a visiting student at the Information Systems Group, led by Prof. Felix Naumann.

Entity Resolution is a major aspect of Data Cleaning and Data Preparation. Nowadays, it is more important than ever considering the massive amount of data, mostly from the web, we deal with every day. It is the Big Data era. Algorithms have been developed during the years in order to shorten the time a system needs to clean all its data before using them. Progressive algorithms are fundamental to achieve the goal of efficiency: those algorithms are able to resolve entities (to identity records that refer to the same real-world object and merge them together) progressively, without waiting the end of the cleaning of the entire dataset. The novelty I introduce with this thesis is to point the Entity Resoution process only to the data I need to achieve the goal. In particular, I created a Query-driven approach that aims to order clean data according to a query the user had set. The records needed are progressively cleant and ordered and the results are given with less amount of comparisons and therefore computational effort than using any other cleaning algorithm.

Sommario

Questa tesi è il risultato di una ricerca sviluppata in collaborazione con il Hasso-Plattner Institut di Postdam, in Germania. In particolare, ho trascorso due mesi come visiting student presso l' Information Systems Group, guidato dal prof. Felix Naumann.

Entity Resolution è un aspetto principale del Data Cleaning e Data Preparation. Oggigiorno, è ancora più importante considerando la quantità di dati, prevalentemente dal web, con cui lavoriamo quotidianamente. E' l'era dei Big Data. Nel corso degli anni sono stati sviluppati algoritmi con lo scopo di diminuire il tempo richiesto al sistema per pulire tutti i dati al suo interno prima di poterli utilizzare. Gli algoritmi progressivi sono fondamentali per raggiungere l'obiettivo di efficienza: questi algoritmi sono capaci di risolvere entità (identificare record che riferiscono allo stesso oggetto del mondo reale ed unirli) progressivamente, senza aspettare la fine della pulizia completa del dataset. La novità che introduco con questa tesi è di indirizzare il processo di Entity Resolution solamente verso i dati che occorrono ad un preciso scopo. In particolare, ho creato un approccio Query-driven che punta ad ordinare i dati puliti secondo una query impostata dall'utente. I record che occorrono sono progressivamente puliti ed ordinati e i risultati sono ottenuti con un numero molto minore di comparison e quindi con un costo computazionale più ristretto rispetto qualsiasi altro algoritmo di cleaning.

Contents

Chapter 1

Introduction

Knowledge is the most precious resource for an organization. In fact, knowledge is part of what distinguishes a company from a competitor of the same sector, e.g. know-how of certain processes and their management, creation of competitive products, etc. The knowledge that one possesses is expressed in the form of data, using them properly is crucial. Modern companies have potentially access to unlimited data sources, e.g. web data folders, posts on social media, stream data from web portals, etc. Analysts usually want to integrate multiple sources together to perform analysis and to support decision making. For example, compare the prices of multiple online stores. As a result of merging data from different sources, a given real-world object can have multiple representations (i.e. different records). This generates data quality problems and could have negative economic impacts.

In this thesis, we focus on the Entity Resolution (ER) challenge, the task of which is to discover duplicate entities that refer to the same real-world object and then to group them into a single cluster that uniquely represents that object.

Traditionally, entity resolution, and data cleaning in general, is performed in the context of data warehousing as an offline preprocessing step prior to making data available to analysis – an approach that works well under standard settings. Such an offline strategy, however, is not viable in emerging applications that deal with big data analysis. First, the need for (near) real-time analysis requires modern applications to execute up-to-the-minute analytical tasks, making it impossible for those applications to use time-consuming standard back-end cleaning technologies. Another reason is that in the data analysis scenarios that motivate our work, an analyst/user may discover and analyze data as part of a single integrated step. In this case, the system will know “what to clean” only at analysis time (while the user is waiting to analyze the data). Last, given the volume and the velocity of big data, it is often infeasible to expect that one can fully collect or clean data in its entirety.

1.1 Query-Driven Approach

Query-Aware ER is an emerging paradigm for data cleaning that supports the growing need for data analysis applications that run in near real-time.

We focus on developing a method that could directly apply a database query on a generic dataset, which is not organized in tables. With this, we allow the user to formulate a query to retrieve some information as he would do with a more structured set of data.

In this thesis we propose an algorithm that, taken as input an SQL-like query entered by the user, interrogates the dataset and reports the results ordered by relevance, satisfying the *ORDER BY* clause. In particular, the algorithm focuses on numerical attributes for sorting entities. Below is an example of a query to which the algorithm answers:

```
SELECT id, min(Price) as mp
FROM Smartphones
GROUP BY _
ORDER BY mp ASC
```

In the query above, data are queried as if they were in a structured database. The query asks for the cheapest product of each group of smartphones. The entity resolution aspect relies on the *GROUP BY* clause and by this, records are merged according to the entity they refer to. The novelty though, is the *ORDER BY* clause, which is the fundamental element of our research. In fact, our main goal is to provide an algorithm able to sort large amount of data in short time and with less comparisons than traditional methods.

By this, we aim to clean, i.e. merging duplicates, only the portion of data that is needed for the query. There is no need for the algorithm to resolve all the records in the dataset while only a small portion contains what the user asks for.

Once accomplished the first query, we decided to extend the algorithm to more complex queries such as the following:

```
SELECT id, min(Price) as mp
FROM Smartphones
GROUP BY _
HAVING Name LIKE "%i-Phone%"
```

```
ORDER BY mp ASC
```

As we notice, we added the $HAVING$ clause here. We will describe later in Chapter 3 the details but with this we allow the user to be more specific with his requests.

1.2 Organization

This thesis addresses the Data Integration challenge, in particular aims to answer the unresolved problem of sorting large amount of data coming from dirty sources, in a progressive way.

The organization of the thesis is the following:

- Chapter 2 depicts the theory and the basics of Data Integration. Some definitions will be described, along with modern methods already implemented in the field of entity resolution.
- Chapter 3 contains the description of the algorithm we created and tested. The details and the novelty that we bring to the research.
- Chapter 4 illustrates the experiments that we have done to test our algorithm and the results we have achieved.
- Chapter 5 finally summarises the accomplishments of this research and opens for future developments.

Chapter 2

Related Work

Big data is being generated and used today in a variety of domains, including data-driven science, telecommunications, social media, large-scale e-commerce, medical records and e-health, and so on. Since the value of data explodes when it can be linked and fused with other data, addressing the Big Data Integration (BDI) challenge is critical to realizing the promise of big data in these and other domains. As one prominent example, recent efforts in mining the web and extracting entities, relationships, and ontologies to build general purpose knowledge bases show promise of using integrated big data to improve applications such as web search and web-scale data analysis. As a second important example, the flood of geo-referenced data available in recent years, such as geo-tagged web objects (e.g., photos, videos, tweets), online check-ins, WiFi logs, GPS traces of vehicles and roadside sensor networks has given momentum for using such integrated big data to characterize large-scale human mobility and influence areas like public health, traffic engineering, and urban planning.

In this chapter we aim to introduce important concepts that have been using to accomplish this research. First of all, elements of theory and knowledge from literature will be presented as the fundamental mile-stones of Data Management. We describe definitions such as Big Data Integration, Entity Resolution, entity resolution and different techniques that have been developed to address the problem of Data Cleaning.

2.1 Data Redundancy

Data integration (DI) has the goal of providing unified access to data residing in multiple, autonomous data sources. While this goal is easy to state, achieving this goal has proven notoriously hard, even for a small number of sources that provide structured data.

The data obtained from different sources often overlap, resulting in a high data redundancy across the large number of sources that need to be integrated. One key advantage of this data redundancy is to effectively address the veracity challenge in DI, as we discuss in detail in the next sections. Intuitively, if there are only a few sources that provide overlapping information, and the sources provide conflicting values for a particular data item, it is difficult to identify the true value with high confidence. But with a large number of sources, as is the case in DI, one can use sophisticated data fusion techniques to discover the truth.

A second advantage of data redundancy is to begin to address the variety challenge in DI, and identify attribute matchings between the source schemas, which are critical for schema alignment. Intuitively, if there is significant data redundancy in a domain, and the graph of entities and sources is well connected, one can start with a small seed set of known entities, and use search engine technology to discover most of the entities in that domain. When these entities have different schemas associated with them in the different sources, one can naturally identify attribute matchings between the schemas used by the different sources. A third advantage of data redundancy is the ability to discover relevant sources in a domain, when sources are not all known a priori. The key intuition again is to take advantage of a well-connected graph of entities and sources, start with a small seed set of known entities, and use search engine technology to iteratively discover new sources and new entities, in an alternative manner.

2.2 Entity Resolution

As we stated that data redundancy can help identify entities in uncertain situations, it is true that having many records that refer to the same real-world object is a waste of memory and, especially in data science, a waste of computational effort. Therefore, entity resolution techniques are required.

Given a set of records, the objective in entity resolution (ER) is to find clusters of records such that each cluster collects all the records referring to the same entity. For example, if the records are restaurant entries on Google Maps, the objective of ER is to find all entries referring to the same restaurant, for every restaurant. In many cases, the popularity of different entities is not the same: few entities collect a large number of records referring to them, while for most entities there are is a single or a couple of records referring to them. Moreover, there are many applications that only need to find those few entities with the large number of records referring to them, and do not need all other less popular entities.

We first formally define the problem of entity resolution in data integration. Let $\mathcal{E}$ denote a set of entities in a domain, described using a set of attributes $\mathcal{A}$. Each entity $E \in \mathcal{E}$ is associated with zero, one or more values for each attribute $A \in \mathcal{A}$. We consider a set of data sources $\mathcal{S}$. For each entity in $\mathcal{E}$, a source $S \in \mathcal{S}$ provides zero, one or more records over the attributes $\mathcal{A}$, where each record provides at most one value for an attribute. We consider atomic values (string, number, date, time, etc.) as attribute values, and allow multiple representations of the same value, as well as erroneous values, in records. The goal of entity resolution is to take the records provided by the sources as input and decide which records refer to the same entity.

Entity Resolution definition. Consider a set of data sources $\mathcal{S}$, providing a set of records $\mathcal{R}$ over a set of attributes $\mathcal{A}$. Entity Resolution computes a partitioning $\mathcal{P}$ of $\mathcal{R}$, such that each partition in $\mathcal{P}$ identifies the records in $\mathcal{R}$ that refer to a distinct entity.

Entity Resolution consists of three main steps: blocking, pairwise matching, and clustering. We will describe each of these steps in more detail next, but it is worth keeping in mind that pairwise matching and clustering are used to ensure the semantics of entity resolution, while blocking is used to achieve scalability.

2.2.1 Pairwise matching

The basic step of entity resolution is pairwise matching [12], which compares a pair of records and makes a local decision of whether or not they refer to the same entity. A variety of techniques have been proposed for this step.

Rule-based approaches. These approaches apply domain knowledge to make the local decisions. The advantage of this approach is that the rule can be tailored to effectively deal with complex matching scenarios. However, a key disadvantage of this approach is that it requires considerable domain knowledge as well as knowledge about the data to formulate the pairwise matching rule, rendering it ineffective when the records contain errors.

Classification-based approaches These approaches have also been used for this step, wherein a classifier is built using positive and negative training examples, and the classifier decides whether a pair of records is a match or a non-match; it is also possible for the classifier to output a possible-match, in which case the local decision is turned over to a human. Such classification-based machine learning approaches have the advantage that they do not require significant domain knowledge about the domain and the data, only knowledge of whether a pair of records in the training data refers to the same entity or not. A disadvantage of this approach is that it often requires a large number of training examples to accurately train the classifier.

DISTANCE-BASED APPROACHES These approaches apply distance metrics to compute dissimilarity of corresponding attribute values (e.g., Euclidean distance for computing dissimilarity of numeric attributes), and take the weighted sum as the record-level distance. Low and high thresholds are used to declare matches, non-matches and possible matches. A key advantage of this approach is that the domain knowledge is limited to formulating distance metrics on atomic attributes, which can be potentially reused for a large variety of entity domains. A disadvantage of this approach is that often requires careful parameter tuning (e.g., what should the weights on individual attributes be for the weighted sum, what should the low and high thresholds be), although machine learning approaches can often be used to tune the parameters in a principled fashion.

2.2.2 Clustering

The local decisions of match or non-match made by the pairwise matching step may not be globally consistent. For example, there is an inconsistency if pairwise matching declares that record pair $\mathcal{R}_1$ and $\mathcal{R}_2$ match, record pair $\mathcal{R}_2$ and $\mathcal{R}_3$ match, but record pair $\mathcal{R}_1$ and $\mathcal{R}_3$ do not match. In such a scenario, the purpose of the clustering [12] step is to reach a globally consistent decision of how to partition the set of all records such that each partition refers to a distinct entity, and different partitions refer to different entities. This step first constructs a pairwise matching graph $\mathcal{G}$, where each node corresponds to a distinct record $R \in \mathcal{R}$, and there is an undirected edge $(\mathcal{R}_1\ , \mathcal{R}_2\ )$ if and only if the pairwise matching step declares a match between $\mathcal{R}_1$ and $\mathcal{R}_2$. A clustering of $\mathcal{G}$ partitions the nodes into pairwise disjoint subsets based on the edges that are present in $\mathcal{G}$. There exists a wealth of literature on clustering algorithms for entity resolution. These clustering algorithms tend not to constrain the number of clusters in the output, since the number of entities in the data set is typically not known a priori.

One of the simplest graph clustering strategies efficiently clusters graph $\mathcal{G}$ into connected components by a single scan of the edges in the graph. Essentially, this strategy places a high trust on the local match decision, so even a few erroneous match decisions can significantly alter the results of entity resolution.

At the other extreme, a robust but expensive graph clustering algorithm is correlation clustering . The goal of correlation clustering is to find a partition of nodes in $\mathcal{G}$ that minimizes disagreements between the clustering and the edges in $\mathcal{G}$, as follows. For each pair of nodes in the same cluster that are not connected by an edge, there is a cohesion penalty of 1; for each pair of nodes in different clusters that are connected by an edge, there is a correlation penalty of 1. Correlation clustering seeks to compute the clustering that minimizes the overall sum of the penalties (i.e., the disagreements).

2.2.3 Blocking

Pairwise matching and clustering together ensure the desired semantics of entity resolution, but may be quite inefficient and even infeasible for a large set of records. The main source of inefficiency is that pairwise matching appears to require a quadratic number of record pair comparisons to decide which record pairs are matches and which are non-matches. When the number of records is even moderately large (e.g., millions), the number of pairwise comparisons becomes prohibitively large. Blocking was proposed as a strategy to scale entity resolution to large data sets.

The basic idea is to utilize a blocking [12] function on the values of one or more attributes to partition the input records into multiple small blocks, and restrict the subsequent pairwise matching to records in the same block. The advantage of this strategy is that it can significantly reduce the number of pairwise comparisons needed, and make entity resolution feasible and efficient even for large data sets. The disadvantage of this strategy is false negatives: if there are incorrect values or multiple representations in the value of any attribute used by the blocking function, records that ought to refer to the same entity may end up with different blocking key values, and hence could not be discovered to refer to the same entity by subsequent pairwise matching and clustering steps.

The key to addressing this disadvantage is to allow multiple blocking functions [Hernández and Stolfo, 1998]. As shown in Figure 2.2, using multiple blocking functions could result in high quality entity resolution without necessarily incurring a high cost. In general, such blocking functions create a set of overlapping blocks that balance the recall of entity resolution (i.e., absence of false negatives) with the number of comparisons incurred by pairwise matching.

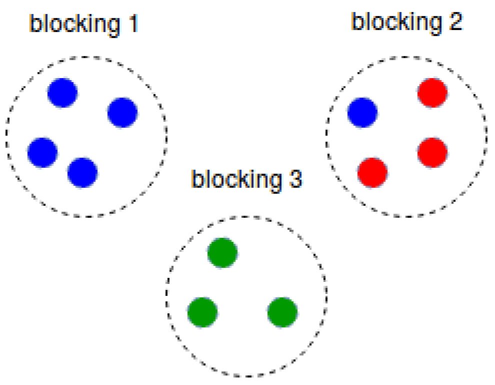

Figure 2.1: Single function Blocking

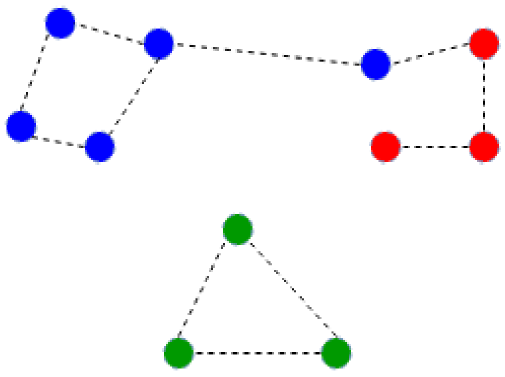

Figure 2.2: Multiple function Blocking

2.2.4 Clean-Clean and Dirty ER

At the core of ER lies the notion of entity profile (or simply profile), which constitutes a uniquely identified set of attribute name-value pairs. An individual profile is denoted by p i , with i standing for its id in a profile collection P . Two profiles $\mathcal{P}_i$, $\mathcal{P}_j \in$ P are called duplicates or matches ($\mathcal{P}_i \equiv \mathcal{P}_j$) if they represent the same real-world entity. Depending on the input data, ER takes two forms:

1. *Clean-Clean ER* receives as input two duplicate-free, but overlapping profile collections, $\mathcal{P}_1$ and $\mathcal{P}_2$, and returns as output all pairs of duplicate profiles they contain, $\mathcal{P}_1 \cap \mathcal{P}_2$.

2. *Dirty ER* takes as input a single profile collection that contains duplicates in itself and produces a set of equivalence clusters, with each one corresponding to a distinct profile.

2.3 Query-Aware ER

Query-aware cleaning is emerging as a new paradigm for data cleaning to support today's increasing demand for (near) real-time analytical applications. Modern enterprises have access to potentially limitless data sources, e.g., web data repositories, social media posts, clickstream data, etc. Analysts usually wish to integrate one or more such data sources (possibly with their own data) to perform joint analysis and decision making. As a result of merging data from different sources, a given real-world object may often have multiple representations, resulting in data quality challenges.

A query-driven approach is motivated by several key perspectives. First, the need for (near) real-time analysis requires modern applications to execute up-to-the-minute analytical tasks, making it impossible for those applications to use time-consuming standard back-end cleaning technol- ogies. Second, in the case of data analysis scenario (e.g., queries on online data), where a data analyst may discover and analyze data as part of a single integrated step, the system will know "what to clean" only at query time (while the analyst is waiting to analyze the data).

We analyze two techniques [5][8] that have been implemented to address this cleaning challenge.

- *QuERy.* The adaptive cost-based approach of this solution, given a query tree (with polymorphic operators) and dirty entity-sets, can devise a good plan to simultaneously clean and process the query. The key intuition hinges on placing decision nodes as the bottommost nodes in the query tree. The task of such decision nodes is to decide if eagerly cleaning some dirty blocks is more efficient (in terms of the overall query execution time) than delaying their cleansing until the last stage as in the lazy solution. The conjecture of placing these nodes at the bottom is to allow adaptive-QuERy to make the "cleaning a block eagerly versus passing it up the tree" decision, from the start of query execution time. This adaptive cost-based solution consists of two steps. In the first step, we use a sampling technique to collect different statistics (e.g., selectivities of predicates, cost of join, etc.). In the second step, the decision nodes utilize these statistics to make their smart decisions.

- *QDA: Query-Driven approach.* Given a block B, and an arbitrary complex selection predicate $\mathcal{P}$, QDA analyzes which entity pairs do not need to be resolved to identify all entities in $\mathcal{B}$ that satisfy $\mathcal{P}$. It does so by modeling entities in $\mathcal{B}$ as a graph and resolving edges (potentially) belonging to cliques that may change the query answer. QDA computes answers that are equivalent to those obtained by first using a regular cleaning algorithm, and then querying on top of the cleaned data. However, in many cases QDA computes such answers much more efficiently. A key concept driving QDA is that of vestigiality. A cleaning step (i.e., call to resolve) is vestigial (i.e., unnecessary) if QDA can guarantee that it can still compute a correct final answer without knowing the outcome of this resolve.

As a result, QuERy operates at macro (block) level: should a block be cleaned or not. In contrast, QDA aims to reduce the number of cleaning steps that are necessary to exactly answer selection queries spanning a single dirty relation. In particular, it proposes algorithms for cleaning entities within a block. It operates at micro (entity pair) level: should an entity pair inside a block be resolved or not. Note that, in theory, QuERy could leverage QDA to be even more efficient by exploiting vestigiality analysis from the latter at the block level to reduce the number of entity pairs that are resolved within a block.

2.4 Progressive Entity Resolution

Alongside detecting duplicates correctly, which is a crucial aspect of nowadays business which deal with large amount of data, another fundamental feature to keep under control is efficiency and, in particular, the time needed to perform data cleansing.

Progressive entity resolution identifies most duplicate pairs early in the detection process compared with traditional methods. Instead of reducing the overall time needed to finish the entire process, progressive approaches try to reduce the average time after which a duplicate is found. Early termination, in particular, then yields more complete results on a progressive algorithm than on any traditional approach.

2.4.1 PSNM

Progressive Sorted Neighborhood (PSN) [17] is the fundamental step towards the direction of saving time and cost. Based on Batch Sorted Neighborhood, it associates every profile with a schema-based blocking key. Then, it produces a sorted list of profiles by ordering all blocking keys alphabetically. Comparisons are progressively defined through a sliding window, w, whose size is iteratively incremented: initially, all profiles in consecutive positions (w=1) are compared, starting from the top of the list; then, all profiles at distance w=2 are compared and so on and so forth, until the processing is terminated. The Neighbor List can be built from schema-based or from schema-agnostic blocking keys and is typically employed to generate blocks: a window slides over the Neighbor List, and blocks correspond to groups of profiles that fall into the same window. The size of the window is iteratively incremented.

The resulting blocks are called redundancy-neutral blocks, because the similarity of two profiles is not related to the number of blocks they share; the corresponding blocking keys might be close when sorted alphabetically, but rather dissimilar. However, the performance of PSN depends heavily on the attribute(s) providing the schema-based blocking keys that form the sorted list(s) of profiles. In case of low recall, the entire process is repeated, using multiple blocking keys per profile. As a result, PSN requires domain experts, or supervised learning on top of labeled data in order to achieve high performance. In contrast, our methods are completely unsupervised and schema-agnostic. PSNM sorts the input data using a predefined sorting key and only compares records that are within a window of records in the sorted order. The intuition is that records that are close in the sorted order are more likely to be duplicates than records that are far apart, because they are already similar with respect to their sorting key. More specifically, the distance of two records in their sort ranks (rank-distance) gives PSNM an estimate of their matching likelihood. The PSNM algorithm uses this intuition to iteratively vary the window size, starting with a small window of size two that quickly finds the most promising records. This static approach has already been proposed as the Sorted List of Record Pairs hint [3]. The PSNM algorithm differs by dynamically changing the execution order of the comparisons based on intermediate results (Look-Ahead).

Chapter 3

Algorithm

In this section, the OrdER algorithm [18] is explained. The goal is to give a positive solution to the problem described above: to order by relevance the objects contained in a set of data, given a query by the user. The algorithm does not improve the total deduplication time of the entire dataset, but emits results progressively as the entities are identified. The user will not need to wait the dataset to be completed clean before getting the first ordered objects.

3.1 General description

The OrdER algorithm aims to clean data and order objects to give them to the user as soon as possible. The order by which the results are given is final: we are sure that there won't be any changes in the sequence. Furthermore, we are sure that the order is correct.

The challenge we want to address is the following: enable users to get information from a generic dataset the same way they would do with a database. Practically speaking, users can insert parameters representing the clauses of an SQL-like query as input and get an appropriate answer. The system then properly reads the parameters, extracts information from the dataset and gives results.

The first step towards this goal is to answer a query like the one below:

```
SELECT Name, MIN(Price) as minimum_price
FROM Smartphones
WHERE Name LIKE '%i-Phone%'
GROUP BY _
ORDER BY minimum_price DESC
```

Let's analyze the query one line at a time. The user who inserted this query would like to get some information about smartphones, as we can see in the clause *FROM Smartphones*. In particular, the clause *WHERE Name LIKE '%i-Phone%'* says that the name of each product must contain the string 'i-Phone', so we are narrowing the range of the objects we want to deal with. Specifically, we are filtering all the records in the dataset and we will not consider those that don't match the clause. The *GROUP BY* clause is the key-step in the duplicate detection process: as expressed like in the query above, it indicates that the records must be grouped by the entity they refer to. In this example, we may have many records referring to the same i-Phone and we want to resolve them into one. How do we accomplish this? This is where the *SELECT* clause plays its part: we want to get the name and the price of the i-Phones with the minimum price in the dataset.

Let's consider an example: we want to find the cheapest of every model of i-Phone at an online shop. It may happen that the online shop just integrated few datasets from previous competitors, which have been bought, in its first dataset. Now we have more records referring to the same products and since they came from different sources, they also have different prices. What is requested is for every i-Phone model, to gather all the records from all the sources and among all those, find the one with the minimum price.

Finally, we want to order all the i-Phone entities (i.e., the records in which all the others have been merged) by the minimum price in ascending order.

The OrdER algorithm enables the user to get results progressively in short time. This is important as we stated that nowadays businesses may not have the time to wait the clean process over all records. One of the main achievement of OrdER is that the user can stop the algorithm at time t as $t < T$ where T is the amount of time required to clean and order all the data. By time t OrdER has provided the user with the most relevant records.

3.2 Query-Driven ER

3.2.1 Modified PSN

We described Progressive Sorted Neighborhood Method [2] in section 2 and we stated how earlier user could get first results compared to standard approaches. OrdER algorithm implements a modified version of PSN. PSNM is meant to be a duplicate detection algorithm while OrdER's first goal is ordering entities correctly as duplicate detection is only a step to achieve that.

PSNM aims to resolve as many duplicates as possible comparing pairs at a certain distance in the sorted list and changing this distance until a fixed window size is reached. Our version of PSN assumes that similar records are likely to be close to one another in SL list and that there are entities which contain more elements than others. Therefore, our PSN searches for comparisons between a record i and its neighbors only within a window whose size is fixed and if a match with neighbor j occurs, records i and j are saved in a partially constructed entity. Afterwards, the algorithm "follows the match", i.e. repeats the same procedure in the neighborhood of j and so forth. The modified PSN stops when no more matches are found and the records collected are marked as resolved.

3.2.2 OrdER Duplicate Detection

The main idea of the algorithm is to implement two arrays to be used to clean while ordering. The first list is called Ordering List (OL) and it contains all the records we are considering and then is sorted by the $ORDER\ BY$ clause the user has inserted. So following the example in the previous section, the OL array is sorted by Price in ascending order. By now though, the records are not resolved and OL may contain duplicates. This is why we need a second list, which is called Solving List (SL) and it also contains all the records. This list is used for Duplicate Detection and resolving entities. In particular, SL is sorted by a solving-key the user inserts in the system. The solving-key must be chosen in order to make a sorted array which is likely to have similar records close to each other.

The algorithm starts from the first element in the OL list and proceeds to the last, one step at a time.

1. Identify next record's id.
2. Look up in the SL list for the same id.
3. Compare record with neighbors on both side in a range defined by window size.
4. If a neighbor matches, compare it with its own neighbors in the same range. Do it recursively.
5. Apply the merging function to the values collected from the matching records. The result will be the value of the entity. The greatest among all the entities' value is the bound (descendant order).
6. Marked all matching records as solved.

7. If the next record's value in OL is lower than the bound, emit the entity related to the bound (MIN merging function).

Let's now create an example of OL and SL related to the query in the previous section: the merging function is MIN and the order is descending. We want to order the entities according to their prices. We create OL by sorting the records from the greatest price to the lowest (descending) and SL with a key inserted by the user.

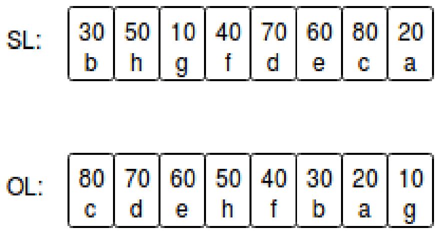

Figure 3.1: OL and SL pt1

Now, we have to resolve entities, which means finding all the records referring to the same object and merge them. In our case, we want the cheapest one so as soon as we identify a group of records, we will keep the one with the lowest price and "erase" the others which are no longer useful.

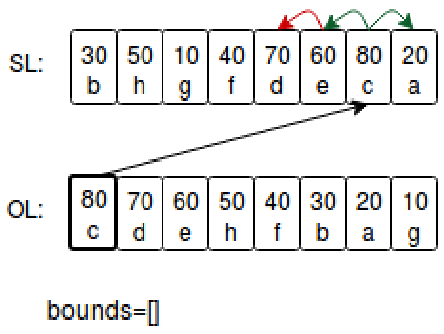

Figure 3.2: OL and SL pt2

Figure 3.2 shows the resolving process of record c: the algorithm identifies the id of record c from the OL list and looks up for it in the SL list. In the SL list, the algorithm starts making comparisons with the neighbors of record c, which are record e and record a. Since they match, the algorithm tries to compare e and a with their neighbors. Only e has a neighbor so it is compared with record d and they don't match. The algorithm stops looking for other matches.

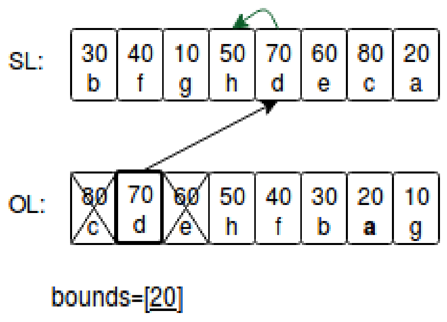

Figure 3.3: OL and SL pt3

Figure 3.3 shows how record a, which has the lowest price, is chosen as representative of the object while the other two are not considered anymore: 20 is the only and actual bound of the algorithm. Now we resolve record d and it is compared only with the neighbor to the left in the SL list since e is already solved. Current

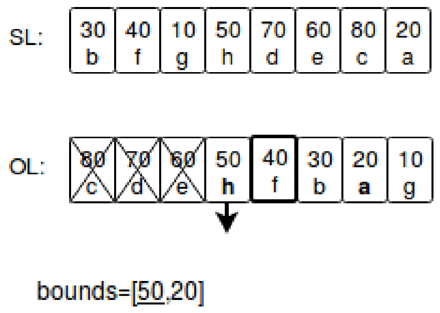

Figure 3.4: OL and SL pt4

Figure 4.4 shows records d and h being resolved as 50 is the entity's minimum price and it goes in the *bounds* list. We already solved h, so we step onto f whose price is 40. Since 40 is lower than the actual bound 50, we emit record h.

3.3 OrdER

Algorithm 1 is the representation of the main part of OrdER and it shows how functions and variables are handled. The requirements are a dataset reference to disk; a solving key used for the Solving List; an ordering key for the Ordering List; a SQL-like merging function; a window size for the modified PSNM and a query order to sort entities.

The algorithm first creates SL and OL arrays giving them the size of the dataset, then it loads the records and sort records' ids into the two arrays according to the keys inserted by the user. SL and OL only take the ids, otherwise they would be too heavy. Now a For-loop begins: each id in the OL list is extracted in an iterative way and the related record as well. In line 11 there is a $While$-loop, valueBeyondBound() is invoked and it checks whether we should emit the next entity or not. Specifically, if the record's value is beyond the current bound, as we stated in the previous section, we emit the entity related to that bound which is always in position 0. We iterate in this loop over again as long as the current record's value steps behind the bound. Line 14 we take the SL's index of the actual id and we give it as input to $computeEntityValue$, along with SL, window size W and merging function MF. If not terminated early by the user, OrdER finishes when all records in OL have been processed.

Algorithm 1 OrdER

Require: dataset reference D, sorting key SK, ordering key OK, merging function MF, window size W, query order O

```
 1: procedure Order($D, SK, OK, MF, W, O$)
 2:     array $OL$ size D.size as Integer
 3:     array $SL$ size D.size as Integer
 4:     $recs \leftarrow$ loadPartition($D$)
 5:     $OL \leftarrow$ orderData($recs$)
 6:     $SL \leftarrow$ sortData($recs$)
 7:     $bounds \leftarrow \{\}$
 8:     for $i = 0$ to $D.size$ do
 9:         $id \leftarrow OL[i]$
10:         $record \leftarrow recs[id]$
11:         while valueBeyondBound($record.value, bounds[0], O$) do
12:             emitEntity($bounds[0]$)
13:         end while
14:         $index \leftarrow$ getPositionSl($id$)
15:         $bounds[id] \leftarrow$ computeEntityValue($SL, index, W, MF$)
16:     end for
```

Algorithm 2 shows function Value Beyond Bound. This function takes three inputs: value of record we are considering at this moment, the bound and the query order. Let's suppose we consider a descending order: the OL list starts with the greatest value and ends with the lowest and since we are resolving each and every record from the beginning, it means that if the next record's value is lower than the actual bound, there will never be another value higher than the bound, thus we should emit that entity.

Algorithm 3 depicts Compare Entity Value which is the part of

Algorithm 2 Value Beyond Bound

Require: value of current record V, upper or lower bound B, query order O

```
1: function valueBeyondBound($V, B, O$)
2:     if $O$ is ascending and $V > B$ then
3:         return $True$
4:     end if
5:     if $O$ is descending and $V < B$ then
6:         return $True$
7:     end if
8:     return $False$
```

the code implementing duplicate detection. In line 2 there is a *For*-loop over the multiple SL lists we may have, there are details about this next section. Line 4, the algorithm iterates neighbors in *dist*-range to pair them with current record. The comparison is executed using the compair(*pair*) function in line 8. If this function returns "true", a duplicate has been found and the lookAhead() method is invoked to progressively search for more duplicates in the current neighborhood.

Algorithm 3 Compare Entity Value

Require: lists of records SL, index of record I, window size W, merging function MF

```
1: function computeEntityValue($SL, I, W, MF$)
2:     for $i = 0$ to $SL.size$ do
3:         $slist \leftarrow SL[i]$
4:         for $dist \in range(-W, W)$ do
5:             $record \leftarrow slist[I]$
6:             $neighbor \leftarrow slist[I + dist]$
7:             $pair \leftarrow \langle$record,neighbor$\rangle$
8:             if $compare$(pair) then
9:                 $values \leftarrow values \cup neighbor$.value
10:                lookAhead($neighbor$,$values$)
11:            end if
12:        end for
13:    end for
14:    $mergedValue \leftarrow$ mergeValues($values$,$mF$)
15:    return $mergedValue$
```

3.4 Multiple Sorted Lists

A fundamental evolution of the algorithm described above is to have *multiple Sorted Lists* to exploit in order to achieve a better duplicate detection. The user has to provide the system, along with the parameters that the algorithm requires, with n keys where $n > 1$. The keys are attributes of the dataset. A sorted list is created from each of these keys by simply sorting by the attribute indicated, whether is ascendant or descendant. The only important thing is that similar items will likely be close to each other. Moreover, by using multiple sorted lists we are increasing the chances to resolve entities completely, i.e., to find all the records representing the same objects (i.e., records that match with each other).

Let's consider an example: we have a set of records $[a, b, c, d, e, f, g, h, i, j, k]$ representing electronic products in a warehouse. The user wants to resolve the entity referred by record a. The user inserted into the system 3 keys by which sorted lists will be created. Let's say the three keys are the following: "maker", "model" and "price". Now we have to find hopefully all the records that match with a. For the sake of the example, suppose we utilize window=1 for our algorithm, that means we only compare one neighbor to the right and one neighbor to the left from record a. If one record matches, it will be compared with its neighbor and so on. From the Ground Truth (i.e., the list of all the matching records) we know that record a matches with records c, e, f. The picture below shows how having multiple sorted lists enhances the efficacy of duplicate detection:

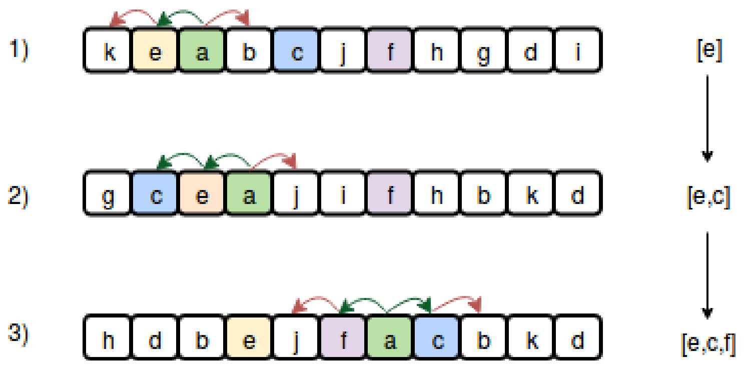

Figure 3.5: Esempio multiple SL

Let's assume the first sorted list uses the "maker" attribute, as we can see it detects record e as a match, while c and f are too far. We save the matching records in a list as we collect them. The second sorted list uses the "model" attribute and finds record c as a match.

Last sorted list sorts records by "price" and it manages to get the last matching record f.

We may wonder why using more than one attribute to create multiple sorted lists when we could simply use attribute "model" to sort the records as the ones that refer to the same products would be one close to another. As we described data integration in the previous sections, we pointed out that due to the heterogeneity of the sources data are taken from, we might find ourselves with plenty of records with different names but all referring to the same real-world objects. Some records may have names shortened, some others may have technical details in the name, etcetera. Therefore, it's impossible to be sure that same objects have same names. This is why we involve multiple sorting lists.

3.5 HAVING clause

In this section, we want to take a step forward into the process of enabling the user to interrogate the dataset like it was a database. The SQL-like query we want to analyze now is a kind of the following:

```
SELECT ID, MIN(Price) as minimum_price
FROM Smartphones
GROUP BY _
HAVING Name LIKE "%i-Phone%"
ORDER BY minimum_price ASC
```

What stands out here is that we changed from -$WHERE\ Name\ LIKE\ "\%i-Phone\%"$- to -$HAVING\ Name\ LIKE\ "\%i-Phone\%"$. This is a key step: with the WHERE clause written this way, we were filtering all records in the dataset before any duplicate detection or ordering. Recall that data integration and data entry may generate inconveniences such as misspelled names. If we accidentally fall in one of these cases, where some i-phones have been wrongly named, by cutting off all the records whose names don't match with "i-Phone", we are losing information and candidates to be the cheapest. Instead, the HAVING clause is applied by definition after the grouping, which means after the entity resolution. Thus, records with misspelled names can be grouped and considered in the entity's price evaluation.

Let's consider the following example of few records:

ID	Maker	Name	RAM	Price
01	Apple	iPhone 8	2GB	500
02	Apple	i-Phone 8 applestore	6GB	800
03	Apple	i_Phone 8	3GB	600
04	Apple	i-Phone8	4GB	650

All records refer to the same product, a smartphone i-Phone 8. Although, as we can see, names differ from each other; the reasons why this can happen are explained in the previous sections. If we kept the WHERE clause as it was, we would get rid of the first and the third products, as the names do not contain the string "i-Phone". In contrast, the new HAVING clause first resolves records and finds whether they refer to the same product and since they do, it groups them and take the ID and the Price of the cheapest one, which is the first.

In Algorithm 1 depicted in section 3.3 we have shown that we start resolving the first element in the Ordering List (OL) and then we proceed to the last one. That is because OL reflects the sorting order of the $ORDER\ BY$ clause. When we add the $HAVING$ clause, the algorithm changes a little.

Strict Bounds. A major change in Algorithm 1 is that OrdER with $HAVING$ clause loses some iterativeness for the sake of completeness. We know which entity has to be emitted only when all the records in the OL, which is smaller now, are checked, although there is no need to resolve all of them.

Let's assume we want to order our records by the maximum value, since the OL list is composed by m elements where $m < n$ (n number of records in dataset), there may be elements whose values are greater than the values of the records we consider. Figure 3.6 shows the OL that contains only those records that meet the clause (the colored ones). It happens when those elements do not meet the $HAVING$ clause.

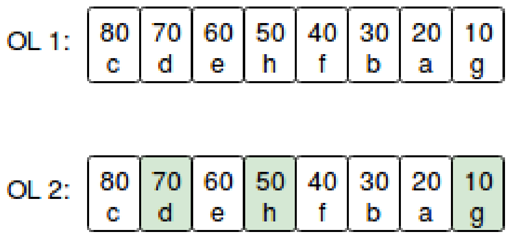

Figure 3.6: OL with HAVING clause

In this situation, for every record in the OL list, we check its neighborhood and if there is at least one record's value which is greater than the actual bound, we resolve the neighborhood and collect the result. If no record's value is bigger than the bound, we simply step to the next element in OL.

Filtered OL. In the OL there are normally all the records of our dataset and they are sorted according to the *ORDER BY* clause. In case of the *HAVING* have to apply a filter on records. We said our algorithm, in contrast with traditional methods, will take in consideration also the elements which don't meet the clause, as long as they match with the ones that do meet the clause. To accomplish this, we mark the records in the dataset that satisfy the *HAVING* clause by checking the value that is involved. With those records which are marked, we build the *OL* list (OL is still sorted) so that as we iteratively take the next element in the list, as shown in Algorithm 1 in Section 3.3, we will only consider the ones which meet the clause.

From this point, in the COMPUTE ENTITY VALUE function, the *SL* list's neighbors of the marked records, which might don't meet the clause, will be taken in consideration for comparisons so they will be not excluded.

Comparing Window Fixed. With the *HAVING* clause, Algorithm 1 changes the way to do comparisons. The window of neighbors in which to seek for matches has to be bigger than the standard one. The reason is that there will be no more "following the match": if a match is found, the comparisons to be made remain the same, no additions.

This is due to improve progressiveness: we said previously that a neighborhood is to be resolved only if there is at least one value higher than the actual bound. To accomplish this properly a set of records has to be defined, this is why a window size fixed.

ALgorithm 4 shows the modifications applied to COMPARE ENTITY VALUE with HAVING clause.

Algorithm 4 OrdER with HAVING clause

Require: dataset reference D, sorting key SK, ordering key OK, merging function MF, window size W, query order O, Having clause H

```
 1: procedure Order($D, SK, OK, MF, W, O, H$)
 2:     array $OL$ size D.size as Integer
 3:     array $SL$ size D.size as Integer
 4:     $recs \leftarrow$ loadPartition($D$)
 5:     $OL \leftarrow$ filterHaving($recs$, $H$)
 6:     $SL \leftarrow$ sortData($recs$)
 7:     $bounds \leftarrow \{\}$
 8:     for each $id \in OL$ do
 9:         $index \leftarrow$ getPositionSl($id$)
10:         for $dist \in range(-W, W)$ do
11:             $neighbor \leftarrow SL[index + dist]$
12:             if $neighbor$.value $> bounds[0]$ then
13:                 $bounds[id] \leftarrow$ computeEntityValue($SL, index, W, MF$)
14:                 break
15:             end if
16:         end for
17:         $record \leftarrow recs[id]$
18:     end for
19:     return emitEntity($bounds[0]$)
```

Chapter 4

Experimental Evaluation

In this section, we evaluate OrdER algorithm on real world datasets and compare it with traditional Sorted Neighborhood and Progressive Sorted Neighborhood. We begin with a description of the setup, the metrics used for comparing our algorithm and eventually the results.

4.1 Metrics

To evaluate our algorithm we needed metrics that could verify whether the results were acceptable or even outperforming traditional SN. Therefore, for all the experiments that we had, three fundamental metrics have been used.

The first one is $Recall$ ($PairCompleteness$ PC), i.e. the completeness of retrieval. For any given retrieved set, Recall is the number of retrieved relevant items as a proportion of all relevant items. Recall is, therefore, a measure of effectiveness in retrieving performance. One-hundred percent Recall can always be achieved by retrieving the entire dataset, but this defeats the purpose of a retrieval system. Dealing with Entity Resolution, PC measures how many duplicates are found, given the total numbers.

The second metric is $Precision$ ($PairQuality$ PQ). For any given retrieved set, Precision is the number of retrieved relevant items as a proportion of the number of retrieved items. Precision is, therefore, a measure of purity in retrieval performance, a measure of effectiveness in excluding nonrelevant items from the retrieved set. In other words, PQ measures how many of the records we found are relevant.

Let ϵ be our dataset:

Where D^d is the number of duplicates found by the algorithm, D^ϵ represents the number of all duplicates in dataset ϵ and C are the comparisons made.

$$PC = \frac{D^d}{D^\epsilon}$$

$$PQ = \frac{D^d}{C}$$

Figure 4.1: Recall and Precision

Last metric used is the F1-score, which is the harmonic mean of Precision and Recall. This is needed when we want to seek a balance between PC and PQ. F1-score summarizes the accuracy of both PC and PQ.

$$F1score = 2 \times \frac{PC \times PQ}{PC + PQ}$$

Figure 4.2: F1 score

Along with the metrics described above, we also measured the values of these features:

- Duplicates
- Entities
- Comparisons

In particular, we want to prove that our algorithm is better at detecting entities with less comparisons than traditional SN.

4.2 Goals

We stated many times that our main goal is to provide an algorithm capable of answer a simple SQL-like query and, in particular, sort entities progressively. In the previous section we introduced the metrics we have been using to judge out algorithm's performance on duplicate detection. In order to evaluate the progressiveness and the efficiency of our algorithm, we estimate Early Ordering Quality and Eventual Ordering Quality. In particular, our goal is to improve the former and preserve the latter.

- Early Ordering Quality: let t be an arbitrary target time at which results are needed. Then our algorithm sorts correctly more entities at t than the corresponding traditional algorithm. Typically, t is smaller than the overall runtime of the traditional algorithm.

- Eventual Quality: if both a traditional algorithm and its progressive version finish execution, without early termination at t, they produce the same results.

Our main objectives confronted with traditional SN are the following:

1. Equal or greater number of entities detected

2. Less comparisons needed to detect the same duplicates, hence the entities

3. Sorting entities correctly and progressively

In particular with *Dirty* datasets, we may find a great number of comparisons in the Ground Truth. For instance, one of our dirty datasets, *CORA*, has 1,879 records and the Ground Truth has 64,578 rows. A traditional algorithm compares all the pairs that is able to reach and might find many matches, scoring a higher Recall than OrdER. OrdER instead, once found that a record belongs to a cluster, it does not compare it with any other record because it would be a waste of time and cost since we already know which entity it represents. This is why in case of GT way bigger than the dataset itself, we look more for the same number of entities compared to Recall.

4.3 Datasets

Our algorithm aims to be useful with all types of situations, especially with everyday problems that many companies have to tackle. One of the issue that often occurs is to merge two datasets from two different sources. This is due to different reasons: company acquiring another and the directive to merge information or an internal re-organizational process whose guideline is to unify same type of data. We assume in these situations that distinct datasets were clean before merging.

The datasets employed as benchmark in the first Clean-Clean Entity Resolution experiments are well known and widely adopted in literature. The first pair of datasets is the *DBLP-ACM*, which contains bibliographic data related to scientific papers:title, authors, venue, year. The second pair is the *DBLP-Scholar* and it collects scientific papers as well. The third pair is the *Amazon-Google Products* and here we find e-commerce products details: name, description, manufacturer, price. The same e-commerce content for the fourth pair: *Abt-Buy*. The last pair of datasets compares again Amazon with another major in the e-commerce business, then we have the *Amazon-Walmart Products*, which has many more attributes than the third pair.

Following is a table depicting most important features have been considered during the experiments.

Feature	Dblp-Acm	Dblp-Sch	Amz-Ggl	Abt-Buy	Amz-Wlmt
Records	4,910	66,879	4,589	2,175	24,630
Sorting key	Title	Title	Model	Model	Model
Ordering key	Year	Year	Price	Price	Price
Matches	2,225	5,348	1,301	1,098	1,155

We have also had experiments on *Dirty* datasets, i.e. where there are duplicates among the records of the datasets themselves. The datasets are three. The first one is called *Affiliation* and contains names of worldwide universities. The second one is the famous *CORA* which incorporates scientific papers details such as author, journal, number of pages etcetera. The third one is the well known and employed *CD* and contains music albums information.

Feature	Affiliation	CORA	CD
Records	2,260	1,879	9,728
Sorting key	Name	Title	Title
Ordering key	Id	Date	Year
Matches	32,817	64,578	299

4.4 Experimental setting and Ground Truth

All experiments have been performed on a system running Ubuntu 16.04, with 8GB RAM and an Intel v7 @ 2.0 GHz CPU. Runtime environment is Python 3.6.

The Ground Truth (GT) for every single dataset we use is well provided. This file contains all the matching pairs between records that can be found in the dataset. It is composed by two columns with the ids of the two datasets analysed, in case of Clean-Clean experiments. When we deal with Dirty datasets on the other hand, we process only one dataset so the records' ids will fill both the columns of GT.

Before executing the experiments, users have to select certain parameters in order to create the SQL-like query that will be applied over the datasets. All the attributes involved are processed to make them suitable for the analysis. In particular, the ids of the Dirty datasets are processed with a label encoder to make them recognizable by the Ground Truth. Moreover, since we deal with numeric attributes for our tests, a cleaning process to remove any other characters is required.

4.5 Results

In this section we illustrate in detail the experiments that have been done, showing how our algorithm outperform traditional SN in many aspects. Following the list of goals presented in the previous section, the first experiments were about resolving as many entities as possible in every dataset. Our intent with this project was not to discover a better entity resolution algorithm so the least goal here is to perform as well as the traditional methods: the *Recall* metric has to be the same or, in case the Ground Truth is larger than the dataset, the exact number of entities has to be found. Additionally, we want the PC to be the same with all the merging functions and the sorting orders we apply.

4.5.1 Clean-Clean ER

The results we declare here are related to those datasets that don't contain duplicates. Only when merged, the problem of duplicate detection arise.

The first outcome we introduce refers to datasets containing scientific papers, *DBLP-ACM* and *DBLP-Scholar*, and to the following query, but the outcome is the same for all other queries:

```
SELECT Title, MAX(Year) as most_recent
FROM D
GROUP BY _
ORDER BY most_recent DESC
```

The ER returns for scientific datasets with a window size of 3 are:

results	SN	OrdER
PC dblp-acm	0.74	0.74
comps dblp-acm	14,724	11,605
PC dblp-scholar	0.60	0.60
comps dblp-scholar	200,631	200,295

The ER returns for scientific datasets with a window size of 20 are:

results	SN	OrdER
PC dblp-acm	0.86	0.86
comps dblp-acm	97,990	63,691
PC dblp-scholar	0.69	0.69
comps dblp-scholar	1,337,370	1,336,805

From the tables we can notice that OrdER has the same Recall but outperforms traditional SN in terms of comparisons as the gain is higher as we increase the window size.

Similar results can be noticed from the following tables, containing the outcome from experiments made with e-commerce products datasets: *Amazon-Google Products*, *Abt-Buy* and *Amazon-Walmart Products*, and this query:

```
SELECT Title, MIN(Price) as cheapest_product
FROM D
GROUP BY _
ORDER BY cheapest_product ASC
```

The ER returns for e-commerce datasets with a window size of 3 are:

results	SN	OrdER
PC abt-buy	0.30	0.30
comps abt-buy	6,513	6,512
PC amz-google	0.30	0.30
comps amz-google	13,760	13,752
PC amz-wlmt	0.26	0.26
comps amz-wlmt	73,878	73,875

The ER returns for e-commerce datasets with a window size of 20 are:

results	SN	OrdER
PC abt-buy	0.60	0.60
comps abt-buy	43,250	43,244
PC amz-google	0.39	0.39
comps amz-google	91,569	91,546
PC amz-wlmt	0.42	0.42
comps amz-wlmt	492,350	492,340

So far we have been pointing out that OrdER's effectiveness in ER is the same as traditional SN with the improvement that OrdER makes less comparisons to obtain same results.

At this point, we represent the major achievement of our algorithm, which is the ability of sorting entities while resolving them. Figure 4.3 depicts the experiment made with $DBLP\text{-}ACM$ dataset, showing how soon our algorithm starts emitting entities, compared with traditional SN whose sorting is final.

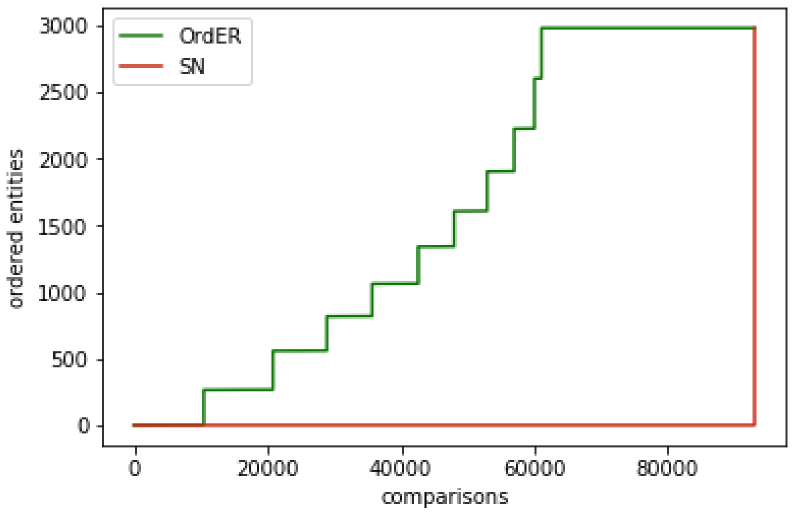

Figure 4.3: OrdER and SN on sorting entities, Clean-Clean

4.5.2 Dirty ER

Now we introduce the achievements obtained with Dirty datasets. Those datasets contain duplicates in their primitive structures, the intention is to merge as many records as possible and sort them. The queries used for the experiments are the same as for the Clean-Clean ones.

The first Dirty datasets employed are $CORA$ and $Affiliation$. For both of them, Ground Truth is bigger than the dataset itself. For this reason in terms of efficacy, Recall is not a reliable metric and instead we rely on the number of entities found. To measure efficiency we still use number of comparisons.

Results with window size of 50:

results	SN	OrdER
Entities CORA	222	223
comps CORA	92,675	14,440
Entities Affil.	762	764
comps Affil.	111,724	39,890

Results with window size of 100:

results	SN	OrdER
Entities CORA	213	214
comps CORA	182,850	23,800
Entities Affil.	696	698
comps Affil.	220,949	68,925

As we can see, OrdER finds two entities less than traditional SN but if we enlarge the window size by one, the results are the same. Plus, OrdER does way less comparisons than SN, which is a big advantage in terms of computational effort.

Figure 4.4 depicts the experiment made with CD dataset, showing how sooner our algorithm starts emitting entities, compared with traditional SN whose sorting is final. The window size used is 100.

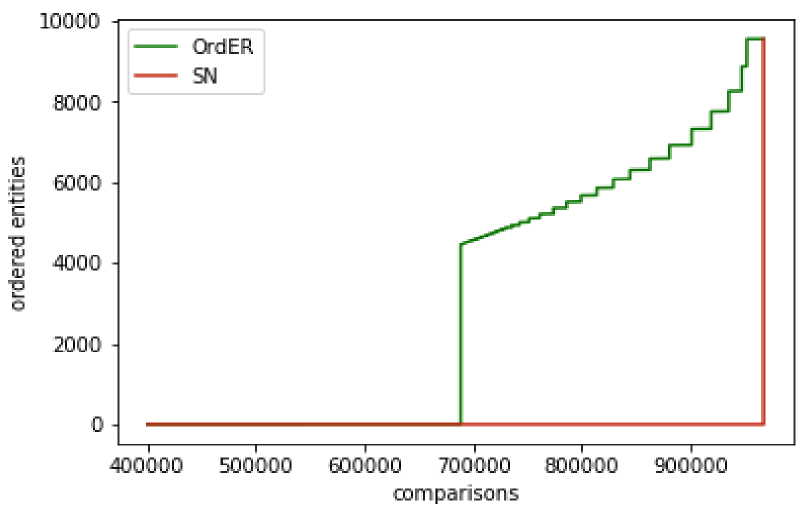

Figure 4.4: OrdER and SN on sorting entities, DirtyER

Chapter 5

Conclusion and Future Work

In this thesis, we have studied the query-driven ER problem in which data is cleaned “on-the-fly” in the context of a selection query. We have developed OrdER, which efficiently issues the minimal number of cleaning steps needed to accurately answer a complex SQL-like query. The main goal of our algorithm was to ensure an accurate sorting of the data according to parameters inserted by the user. We formalized the problem of query-driven ER and showed empirically how certain cleaning steps can be pruned and how our approach is significantly better compared to cleaning the entire dataset, especially when the query is very selective. This research opens several interesting directions for future investigation, e.g. developing solutions to extend our approach as much as possible to not-boundable merging functions such as count or sum.

A further step will be to make the algorithm applicable for very large datasets and, in particular, for those non structured, hence improving the management and manipulation of Big Data.

Bibliography

[1] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool, 2010.

[2] Uwe Draisbach, Felix Naumann, Sascha Szott, and Oliver Wonneberg. *Adaptive windows for duplicate detection*. IEEE 28th International Conference on Data Engineering, 2012.

[3] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. *Pay-as-you-go entity resolution*. IEEE Transactions on Knowledge and Data Engineering, 2012.

[4] I. Bhattacharya and L. Getoor. *Query-time entity resolution*. Journal of Artificial Intelligence Research, 2007.

[5] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. *QDA: A Query-Driven Approach to Entity Resolution*. Proc. VLDB Endowment, 2013.

[6] A.Pietrangelo, G.Simonini, S.Bergamaschi, I. Koumarelas, F. Naumann. *Towards Progressive Search-driven Entity Resolution*. SEDB, 2018.

[7] Chuan Xiao, Wei Wang, Xuemin Lin, Haichuan Shang. *Top- k Set Similarity Joins*

[8] Hotham Altwaijry, Sharad Mehrotra, Dmitri V. Kalashnikov. *QuERy: A Framework for Integrating Entity Resolution with Query Processing.*

[9] Adam Wick. *MAGPIE: PRECISE GARBAGE COLLECTION FOR C*. University of Utah, 2006.

[10] Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, Tova Milo. *A Sample-and-Clean Framework for Fast and Accurate Query Processing on Dirty Data.*

[11] Indrajit Bhattacharya and Lise Getoor. *Collective Entity Resolution in Relational Data*. ACM Transactions on Knowledge Discovery from Data,2007.

[12] Xin Luna Dong, Divesh Srivastava. *Big Data Integration*. Morgan and Claypool Publishers, 2015.

[13] Albert Einstein. *Pay-as-you-go entity resolution* [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891–921, 1905.

[14] Giovanni Simonini, George Papadakis, Themis Palpanas, Sonia Bergamaschi. *Schema-agnostic Progressive Entity Resolution*.

[15] Giovanni Simonini, Sonia Bergamaschi, H.V. Jagadish. *BLAST: a Loosely Schema-aware Meta-blocking Approach for Entity Resolution*.

[16] Thorsten Papenbrock, Arvid Heise, and Felix Naumann. *Progressive Duplicate Detection*.

[17] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano and Sonia Bergamaschi.
`SparkER: Scaling Entity Resolution in Spark`

[18] Giovanni Smonini,
`https://stravanni.github.io/g/progressive_query-driven_ER_technical_report.pdf`

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Hasso-Plattner-Institut für Digital Engineering gGmbH | PF 900460 | 14440 Potsdam

Hasso-Plattner-Institut
für Digital Engineering gGmbH
Campus Griebnitzsee

Postfach 900460
14440 Potsdam

Telefon: +49 (0) 331 5509 -0
Telefax: +49 (0) 331 5509 -129

www.hpi.de

Geschäftsführung
Prof. Dr. Christoph Meinel

Amtsgericht Potsdam
HRB 12184

Potsdam, 21st January, 2019

Research Visit - Giacomo Amici

To Whom It May Concern:

With this letter, I verify that Giacomo Amici was a visiting student at my Information Systems Group at Hasso Plattner Institute from May to July 2018.

If you have any questions regarding Mr. Amici’s visit, I can be reached via email or telephone.

Sincerely,

Prof. Dr. Felix Naumann

Felix.Naumann@hpi.de
+49 331 5509 280