A Software Infrastructure for the CLEENEX Optimizer

Telma Filipa de Carvalho Fernandes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Helena Isabel de Jesus Galhardas

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Helena Isabel de Jesus Galhardas
Member of the Committee: Prof. Pável Pereira Calado

November 2015
Abstract

The problems associated to data quality is an increasingly growing concern. Throughout this document we will focus on a specific data quality problem: the existence of approximate duplicate records. Data cleaning aims at correcting data quality problems that can be found in various situations. There are some data cleaning tools that address these data quality problems. One of the tasks of a data cleaning program consists in the approximate duplicate detection. The approximate duplicate detection must be efficient, because if we are dealing with a large amount of data, comparing all the records will result in a performance bottleneck. The goal of the optimizer in a data cleaning tool is to build several execution plans for the data cleaning program and, based on the cost of each execution plan, choose the most efficient. In order to have the optimizer, we need to build a software infrastructure to support it. In particular, this infrastructure must provide several alternatives that improve the efficiency of the approximate duplicate detection. In this thesis, we designed and implemented an infrastructure to support an optimizer for CLEENEX, a data cleaning tool. In this document we also describe the validation methodology regarding the implemented infrastructure.

Keywords: Approximate duplicate detection, optimizer, data matching, record matching.
Resumo

Os problemas associados à qualidade dos dados é uma preocupação cada vez mais crescente. Ao longo deste documento vamos focar-nos num problema específico de qualidade de dados: a existência de registos aproximadamente duplicados. Um processo de limpeza de dados visa corrigir problemas de qualidade de dados que podem ser encontrados em diversas situações. Existem algumas ferramentas de limpeza de dados que abordam estes problemas de qualidade de dados. Uma das tarefas de um programa de limpeza de dados é a detecção de duplicados aproximados. A detecção de duplicados aproximados deve ser eficiente, porque se estivermos a lidar com uma grande quantidade de dados, comparar todos os registos irá resultar num défice de desempenho. O objectivo do optimizador numa ferramenta de limpeza de dados é a construção de vários planos de execução para o programa de limpeza de dados e, com base no custo de cada plano de execução, escolher o mais eficiente. De modo a ter o optimizador, é necessário construir uma infra-estrutura de software para suportá-lo. Em particular, esta infra-estrutura deve fornecer diversas alternativas que melhorem a eficiência da detecção de duplicados aproximados. Nesta tese desenvolvemos e implementamos uma infra-estrutura para suportar um optimizador para o CLEENEX, uma ferramenta de limpeza de dados. Neste documento, também descrevemos a metodologia de validação tendo em conta a infraestrutura implementada.

Palavras-chave: Detecção de duplicados aproximados, optimizador, data matching, record matching.
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Resumo</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objectives</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Document Structure</td>
<td>5</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Record matching</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 String Matching</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Rule-based Matching</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Scaling Up Rule-based Matching</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 Traditional Blocking</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Sorted-Neighborhood Join</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 Q-gram Based Indexing</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4 Suffix Array Based Indexing</td>
<td>21</td>
</tr>
<tr>
<td>2.2.5 Canopy Clustering</td>
<td>22</td>
</tr>
<tr>
<td>2.2.6 Discussion</td>
<td>24</td>
</tr>
<tr>
<td>3 CLEENEX</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Specification of data cleaning programs</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1 Logical level</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2 Physical level</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Core operation of CLEENEX</td>
<td>26</td>
</tr>
<tr>
<td>3.3 User interaction</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1 Quality Constraints</td>
<td>28</td>
</tr>
<tr>
<td>3.3.2 Manual Data Repairs</td>
<td>29</td>
</tr>
<tr>
<td>3.4 Component Architecture</td>
<td>29</td>
</tr>
<tr>
<td>3.4.1 Parser</td>
<td>30</td>
</tr>
</tbody>
</table>
### 3.4.2 Optimizer

- 3.4.3 Scheduler
- 3.4.4 Debugger

#### 4 Infrastructure for the CLEENEX Optimizer

- 4.1 Solution Overview
- 4.2 Library of External Functions
  - 4.2.1 String matching algorithms
  - 4.2.2 Reorganization of the external functions library
- 4.3 Support for Oracle
  - 4.3.1 Implementation
- 4.4 Scale-up Matching Algorithms
  - 4.4.1 Implementation
- 4.5 Hints
  - 4.5.1 Specification Language
  - 4.5.2 Implementation
- 4.6 Code generation
- 4.7 Integration with the GUI

#### 5 Validation

- 5.1 Experimental Setup
  - 5.1.1 Datasets
  - 5.1.2 Metrics
- 5.2 Scale-up Matching Algorithms
- 5.3 Matching Rules
  - 5.3.1 CDDB
  - 5.3.2 DatasetA
- 5.4 Effectiveness Experimental Results
  - 5.4.1 CDDB
  - 5.4.2 DatasetA
- 5.5 Efficiency Experimental Results
  - 5.5.1 CDDB
  - 5.5.2 DatasetA
- 5.6 Discussion

#### 6 Conclusions

- 6.1 Summary
- 6.2 Future Work

**Bibliography**
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Conversion table used by the Soundex algorithm</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Classification of the scale-up algorithms explained in this Chapter</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Modifications in the classes of the library of external functions</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Type of hints, correspondent values and restrictions</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Number of matches and false matches of the CDDB dataset</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of matches and false matches of the datasetA</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Results after applying the Traditional Blocking to the CDDB dataset</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Results after applying the Traditional SNJ to the CDDB dataset</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Results after applying the Multi-pass to the CDDB dataset</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Results after applying the Inverted Index SNJ to the CDDB dataset</td>
<td>55</td>
</tr>
<tr>
<td>5.7</td>
<td>Best results after applying the Adaptive SNJ to the CDDB dataset</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>Best results after applying the Canopy algorithm to the CDDB dataset</td>
<td>56</td>
</tr>
<tr>
<td>5.9</td>
<td>Results after applying the Traditional Blocking algorithm to datasetA</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>Results after applying the Traditional SNJ algorithm to datasetA</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>Results after applying the Multi-pass algorithm to datasetA</td>
<td>57</td>
</tr>
<tr>
<td>5.12</td>
<td>Results after applying the Inverted Index SNJ algorithm to datasetA</td>
<td>58</td>
</tr>
<tr>
<td>5.13</td>
<td>Best results after applying the Adaptive SNJ algorithm to datasetA</td>
<td>58</td>
</tr>
<tr>
<td>5.14</td>
<td>Best results after applying the Canopy algorithm to datasetA</td>
<td>58</td>
</tr>
<tr>
<td>5.15</td>
<td>Results of the efficiency tests performed by the Cartesian Product for each dataset</td>
<td>59</td>
</tr>
<tr>
<td>5.16</td>
<td>Results after applying the Traditional Blocking to the CDDB dataset</td>
<td>59</td>
</tr>
<tr>
<td>5.17</td>
<td>Results after applying the Traditional SNJ to the CDDB dataset</td>
<td>59</td>
</tr>
<tr>
<td>5.18</td>
<td>Results after applying the Multi-pass to the CDDB dataset</td>
<td>60</td>
</tr>
<tr>
<td>5.19</td>
<td>Results after applying the Inverted Index SNJ to the CDDB dataset</td>
<td>60</td>
</tr>
<tr>
<td>5.20</td>
<td>Best results after applying the Adaptive SNJ to the CDDB dataset</td>
<td>60</td>
</tr>
<tr>
<td>5.21</td>
<td>Best results after applying the Canopy to the CDDB dataset</td>
<td>61</td>
</tr>
<tr>
<td>5.22</td>
<td>Results after applying the Traditional Blocking to DatasetA</td>
<td>61</td>
</tr>
<tr>
<td>5.23</td>
<td>Results after applying the Traditional SNJ to DatasetA</td>
<td>61</td>
</tr>
<tr>
<td>5.24</td>
<td>Results after applying the Multi-pass to DatasetA</td>
<td>62</td>
</tr>
<tr>
<td>5.25</td>
<td>Results after applying the Inverted Index SNJ to DatasetA</td>
<td>62</td>
</tr>
</tbody>
</table>
5.26 Best results after applying the Adaptive SNJ with the simple key to DatasetA. .......... 62
5.27 Best results after applying the Canopy algorithm to DatasetA. ......................... 62

A.1 Results of the Traditional Blocking tests. .......................................................... 76
A.2 Results of the Traditional SNJ tests. ................................................................. 76
A.3 Results of the Multi-pass tests. ......................................................................... 76
A.4 Results of the SNJ Inverted Index tests. .............................................................. 76
A.5 Results of the Adaptive SN tests with attribute author as key. ......................... 77
A.6 Results of the Adaptive SNJ tests with attribute title as key. ............................ 77
A.7 Results of the Canopy tests. ............................................................................. 77
A.8 Results of the Cartesian Product execution. ....................................................... 78
A.9 Results of the Traditional Blocking tests. .......................................................... 78
A.10 Results of the Traditional SNJ tests. ................................................................. 78
A.11 Results of the Multi-pass tests. ....................................................................... 79
A.12 Results of the SNJ Inverted Index tests. .............................................................. 79
A.13 Results of the Adaptive SNJ tests for the first rule with attribute phone as key.  . 79
A.14 Results of the Adaptive SNJ tests for the second rule with attribute phone as key. 79
A.15 Results of the Adaptive SNJ tests for the first rule with attribute name as key. .... 80
A.16 Results of the Adaptive SNJ tests for the second rule with attribute name as key. 80
A.17 Results of the Canopy tests for the first rule. ..................................................... 80
A.18 Results of the Canopy tests for the second rule. .............................................. 80
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of computing the Levenshtein distance between two strings.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Finding the common characters between two strings with the Jaro measure.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of computing the TF and the IDF to be used in the TFIDF algorithm.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of applying the Traditional Blocking to a set of records.</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of applying SNJ to a set of records.</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of applying the q-gram based indexing to two records.</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of applying the suffix-array based indexing to a set of records.</td>
<td>21</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of applying the Canopy Clustering to two records.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of a data cleaning program in CLEENEX.</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Sample of records in relationacidspublication.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>CLEENEX component architecture.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>The new CLEENEX library of external functions.</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Functions definition in a data cleaning program.</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>External functions library categories.</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Matching transformation definition.</td>
<td>47</td>
</tr>
<tr>
<td>4.5</td>
<td>Sequence diagram of the generation of the file for a matching transformation.</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Overall effectiveness of the scale-up matching algorithms and the Cartesian Product.</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>Overall efficiency of the scale-up matching algorithms and the Cartesian Product.</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Organizations rely heavily on data stored in their databases to make decisions. Decisions of good quality must rely on data of good quality. Consider a certain company that sends advertising letters to customers whose information is stored in a table called customer-data. Assume that records $r_1$ and $r_2$ of the table customer-data refer to the same person, but $r_1$ has the value ‘Telma Fernandes’ in the attribute name and $r_2$ has the value ‘Fernandes, Telma’ in the same attribute. Since these two distinct records are referring to the same entity, the corresponding client will receive the same advertising letter twice at home. The existence of approximate duplicate records is a typical example of a data quality problem and must be avoided or corrected in order to minimize the costs associated to the company’s marketing.

Besides the existence of approximate duplicate records, other data quality problems may arise in a database. For example: (i) when data lacks some attribute values (i.e., incompleteness); (ii) when all values are present, but they contain errors (typically due to mistypings) or (iii) when data is inconsistent (e.g., $age = 21$ but $birthday = 27/11/2000$).

Data cleaning aims at correcting data quality problems occurring in a database. A data cleaning process is, typically, modeled as a graph of data transformations. Data transformations are operations that we can apply to data. One type of data transformation is matching, that corresponds to the detection of approximate duplicate records in a database. The detection of approximate duplicates \cite{2,12} and the subsequent elimination are two important tasks of a data cleaning process. In order to detect which records are approximate duplicates, we first need to standardize all records and thus converting some attribute’s values into a specific and uniform format (e.g., dates with the same format). This step is necessary to make the matching transformation more effective. Then, we must detect approximate duplicate records by determining whether two records $a$ and $b$ match. In other words, we want to know if $a$ and $b$ refer to the same real-world entity. This process is also known as record linkage \cite{14}, data matching \cite{7}, or as the merge/purge problem \cite{19}, among others.

The data matching process has two main challenges: effectiveness (i.e., accuracy) and efficiency. It is difficult to match a pair of records accurately because, typically, the records are not exact matches. This happens because records sometimes have typing errors, abbreviations, etc. The following record
matching techniques [11] are examples of solutions to address the problem of accurately determine whether a pair of records is a match:

- **String matching**: Convert each record into a string by concatenating all attribute values. Then, apply a string matching algorithm to the pair of strings. Some examples of string matching algorithms are Levenshtein [11], Jaro-Winkler [37], etc. These algorithms receive two strings \( s_1 \) and \( s_2 \) as input and return a similarity value as output. If this value is greater than a previously defined threshold \( t \), then \( s_1 \) and \( s_2 \) are denoted as matches.

- **Rule-based matching**: Classify two records \( a \) and \( b \) as matches or non-matches based on a set of matching rules that compare attributes of these two records through similarity functions (e.g., string matching functions) and return a similarity value. If this similarity value is greater than a given value \( \beta \), then \( a \) and \( b \) are considered matches.

- **Learning-based matching**: Automatically discover matching rules from a training data set (i.e., a set of pairs of records that are known to be matches and a set of pairs of records that are known to be non-matches) supplied by the user. The basic idea of this approach is to learn matching rules from the training data and then apply those rules to new pairs of records in order to classify them as matches or no matches.

Each comparison of two records has an associated cost. If the number of records to be compared is large, comparing each record with all the others (i.e., computing a Cartesian product) will result in a performance bottleneck. This situation brings us to the second challenge of data matching: efficiency.

In order to efficiently match a very large amount of records, we need to minimize the number of record pairs to be compared. Some optimizations were proposed to reduce this number of comparisons, by avoiding to compare record pairs that do not refer to the same real-world entity. The following techniques [11] are examples of proposed solutions to address the challenge regarding efficiency:

- **Hashing**: Distribute the records by various blocks and compare only the records that are inside the same block. The effectiveness of this technique depends on the hashing function that distributes the records into several blocks. This hashing function must arrange the records into blocks such that records that are likely to match stay in the same block.

- **Sorting**: Use a previously defined key value for each record in order to sort the records and compare only with the \( w - 1 \) records, where \( w \) is a predefined window size. The effectiveness of this technique relies on the chosen key value. This key must be discriminative in order to keep records that are most likely to match, closer to each other, and records that are not matches far away from each other.

- **Indexing**: Index records such that for any given record \( x \), we can quickly find other records that are most likely to match \( x \). One example of an indexing technique is the use of an inverted index table. An inverted index table is a data structure that maps a key value to its corresponding records. In this case, the key value can be an attribute value, and the corresponding records the records that share that same attribute value.
• **Canopies:** Group records into overlapping clusters (i.e., canopies) with the use of cheap similarity functions (e.g., Term Frequency/Inverse Document Frequency (TFIDF) [33] [34] [32]). After this, we only compare the records that are inside the same canopy.

There are some data cleaning tools (e.g., Informatica PowerCenter\(^1\) SAS Data Integration Studio\(^2\)) that are specialized in correcting data quality problems. However, since these tools have a fixed implementation for each logical operator, it is not possible to choose the most appropriate implementation of the approximate duplicate detection process. This may result in a decrease of the performance of the approximate duplicate detection process, specially when we are dealing with a large amount of data. The main problem of most data cleaning tools is the fixed implementation for each logical operator. Due to this, it is not possible to choose the most appropriate approach to perform the approximate duplicate detection.

CLEENEX \(^3\) is a data cleaning prototype that enables the separation between the logical specification of a data cleaning process and its physical implementation. At the logical level the user writes the data cleaning program using a language that extends SQL. In the data cleaning program he/she specifies which data transformations should be applied to the dataset. At the physical layer it is defined how those data transformation should be implemented in order for CLEENEX to generate the corresponding code. One way to be able to choose the most appropriate implementation of the approximate duplicate detection is with the use of an optimizer.

The optimizer must choose automatically the best algorithm to scale-up data matching, depending on the logical specification of the data transformation and on the dataset that is being analyzed. This optimizer is inspired in a Relational Database Management System (RDBMS) optimizer, that attempts to determine the most efficient way to execute a given query by considering all possible query plans. In the case of a data cleaning tool, the optimizer should build several execution plans for the data cleaning program and, based on the cost of each execution plan, choose the most efficient. The current data cleaning tools do not have an optimizer that chooses the most appropriate algorithm to implement the approximate duplicate detection.

In order to have the optimizer that chooses automatically the best algorithm to scale-up data matching, we need to build a software infrastructure that supports it. In particular, this infrastructure must provide several algorithms to scale-up data matching in order to enable different implementations for the approximate duplicate detection.

### 1.1 Objectives

The main goal of this thesis is to build an infrastructure to support an optimizer in CLEENEX \(^3\), a data cleaning prototype. Unlike other data cleaning tools, CLEENEX has a clear separation of the logical specification of the data transformations and their physical implementation. This separation enables to

---

\(^1\)https://community.informatica.com/solutions/pcexpress
\(^2\)http://support.sas.com/software/products/etls/
change between various algorithms to scale-up data matching without affecting the logical specification of the transformation. Thus, guaranteeing an efficient execution of the data cleaning process.

As said earlier in this Chapter, one of the main challenges when performing an approximate duplicate detection is efficiency. CLEENEX offers the matching transformation that corresponds to the duplicate approximate detection process. This transformation receives two input tables and detects the pairs of records that most likely refer to the same real entity. The naive physical implementation of the matching transformation is with the Cartesian Product. The Cartesian Product can be executed both in Java or in SQL. The infrastructure we want to develop enables the choice of alternative algorithms to execute the matching transformation.

We want to provide various algorithms that scale-up the data matching process, as an alternative to the Cartesian Product. Besides these algorithms to scale-up the data matching process, we also want to provide various string matching algorithms that allow the user to create record matching rules (i.e., rule-based matching) that are better suited to each type of dataset.

This infrastructure will allow the choice of alternative algorithms to implement the match operator through hints. A hint is an information the user provides when specifying a given matching transformation. The hint is important in order to indicate to the optimizer which algorithm it should choose to implement that matching transformation. The user can also define through hints the parameters (e.g., window size, thresholds, etc) to be used by the algorithm to scale-up data matching. In particular, a parameter that is used by all of these algorithms to scale-up data matching is the key. The algorithms to scale-up data matching use this key in order to group or sort the records according to their key value. The goal is to keep close records that share the same key value. A key can be an attribute field of the record (i.e., simple) or can be formed with parts of one or more attributes of the record (i.e., composed). In order to create composed keys we must implement a generator that creates the keys according to the user specification.

### 1.2 Contributions

The main contribution of this thesis is the creation of an infrastructure to support the optimizer for CLEENEX. This infrastructure consists in:

1. Six string matching algorithms (two sequence-based, two token-based, and two phonetic). These algorithms are available in the external functions library of CLEENEX.
2. A reorganization of the external functions library of CLEENEX. This reorganization consists in grouping the various functions according to their categories.
3. Six algorithms to scale-up data matching. These algorithms are available in the optimizer component of CLEENEX.
4. Hints in the specification language of CLEENEX in order to enable the user to choose which algorithm to scale-up data matching must be chosen. The hints will also allow the user to choose which parameters the chosen algorithm should use.
5. A generator of composed keys. This generator will form keys according to the specification made by the user.

6. A support for the Oracle Relational Database Management System (RDBMS) in order to allow the execution of the Cartesian Product in SQL by a relational RDBMS.

7. The modification of the code generation mechanism in order to implement the match operator according to the chosen algorithm to scale-up data matching.

8. An evaluation of the impact in terms of effectiveness and efficiency of each algorithm to scale-up data matching. This validation allows us to determine in which cases it is better to use a certain algorithm.

1.3 Document Structure

This document is organized into five chapters. Chapter 2 explains the algorithms that address the two challenges in data matching: effectiveness and efficiency. Regarding effectiveness, we explain record matching techniques and regarding efficiency we explain algorithms to scale-up data matching. Chapter 3 presents CLEENEX, the data cleaning tool for which we implemented the infrastructure for the optimizer. Chapter 4 details all the modifications that were performed to CLEENEX in order to build the infrastructure and allow the support for the several algorithms that scale-up data matching. Chapter 5 describes the validation methodology regarding the efficiency and effectiveness for each algorithm to scale-up data matching. Finally, Chapter 6 concludes and presents the future work of this thesis.
Chapter 2

Related Work

As mentioned in Chapter 1, computing a Cartesian product between two tables \( A \) and \( B \) is computationally expensive, since it considers all possible combinations of records from \( A \) and \( B \). One optimization proposed for this method is the use of indexing techniques \([5]\). The idea underlying these techniques consists in partitioning the set of input records into blocks (i.e., blocking \([21]\)), lists or clusters. Only the records that are in the same block are then compared. This way, the number of comparisons of records is reduced. In order to guarantee the most accurate results, we need to assure that the records that are possible matches belong to the same block. To accomplish this, the proposed techniques group the records according to a given criteria (e.g., a key). This key is defined by the user and is composed by one or more attributes of the records.

The comparison of records is an inferential process. The detection of approximate duplicate records by inference is performed through an equational theory that compares each pair of records based on a set of rules that correspond to the situations where those records can be considered equivalent. For example, if we have two records with the same Social Security Number (SSN), the same name but different addresses, we can infer that those records refer to the same person, but he/she has moved.

This Chapter is organized as follows. Section 2.1 describes some record matching techniques that are used to guarantee accurate results in the data matching process. Section 2.2 presents different techniques to efficiently improve the data matching process.

2.1 Record matching

This Section describes some record matching techniques that are used in data matching (i.e., determining whether two records refer to the same entity, as defined in Chapter 1). The main goal of these algorithms is to determine whether a pair of records are matches or not.

One of the most basic approaches on record matching is the use of string matching algorithms. Each of these algorithms is commonly used as a similarity metric to determine whether two strings are similar. One way to use these string matching algorithms in record matching is, for example, concatenate each field of a given record, resulting in a string. This string will be used by a string matching algorithm to
compare with strings resulting from other records. In this approach, two records are matches if their resulting strings produce a value above a certain threshold when applying a string matching algorithm.

However, this approach is not commonly used, since the results are not the more accurate. Instead, these string matching algorithms are used in rule-based matching [5]. This approach classifies records into matches or non-matches by applying a set of rules. These rules are built with a set of similarity measures (e.g., string matching algorithms), that compare certain attributes whose values can help us determine whether the correspondent record belongs to the same real-world entity.

Besides from these approaches, there are other techniques that consist in probabilistic approaches [28] [29]. In these techniques, the decisions are made according to a set of variables over a probability distribution [11]. A variable can be, for example, whether the value of the attribute name of the records \( r_1 \) and \( r_2 \) match. Based on this variable, and having a previous knowledge whether \( r_1 \) and \( r_2 \) really match, we can make matching decisions for other records that comply the same situation.

In Section 2.1.1 we detail several string matching algorithms that compute the similarity between two strings. In Section 2.1.2 we explain the Rule-based Matching, which employs matching rules to classify a record pair as match or non-match.

### 2.1.1 String Matching

In this Section, we detail some well-known string matching algorithms. Although the purpose is to determine whether two strings are similar, different string matching algorithms take different approaches to compare the two strings. The string matching algorithms can be grouped into different categories (e.g., sequence-based, token-based or phonetic).

#### Sequence-based string matching algorithms

These string matching algorithms compute the similarity between two strings based on the sequence of the characters. We describe two common string matching algorithms that use this sequence-based approach.

1) **Levenshtein distance**

   The Levenshtein distance [26] is a simple string matching algorithm that measures the similarity between two strings based on the characters sequence of each string. This algorithm computes the number of modifications (i.e., character substitutions, insertions or removals) that must be made in order to convert one string into the other.

   In Figure 2.1 we have an example of computing the Levenshtein distance between the strings 'filipa' and 'philippa'. In order to initialize, this string matching algorithm builds a matrix where each line corresponds to each character of one of the strings and each row corresponds to each character of the other string. The first step is to fill the first row and the first column with numbers from 0 to the size of the string, as we can see in Figure 2.1(a).

   The remaining positions of the matrix are filled sequentially per row. Each position of the matrix corresponds to a modification. If the character of the row is the same as the character of the column,
then the value that is inserted is the same as in the left-most diagonal of that position, which means that there were no modifications. Otherwise it is inserted the minimum of the number of the above transformations plus 1.

In Equation (2.1) we have the formula that is used to compute the Levenshtein distance. In the first case is when the characters are equal, which mean that there will be no modifications. In the second case is when the characters are different and need to be substituted. In the third and fourth cases is when one of the strings has one more character or one less character than the other string. In these cases, the character in question must be deleted or inserted from one of the strings.

\[
d(i, j) = \begin{cases} 
  d(i - 1, j - 1) & \text{if } x_i == y_j \\
  d(i - 1, j - 1) + 1 & \text{if } x_i <> y_j \\
  d(i - 1, j) + 1 & \text{if } x_i <> y_j \\
  d(i, j - 1) + 1 & \text{if } x_i <> y_j 
\end{cases}
\] (2.1)

In figure (b) we have the resulting matrix, with the final result (in the gray square) and the path that conducted to that result, in other words the modifications that must be made. As we can see the modifications we have to do is to add a ‘h’ and a ‘p’ to ‘filipa’ or remove the ‘h’ and a ‘p’ from ‘philippa’ and change ‘p’ to ‘f’ from ‘philippa’ or substitute ‘f’ to ‘p’ from ‘filipa’. This modifications will result in a cost of 3 with the Levenshtein distance.

In conclusion, when using the Levenshtein distance, the higher the resulting value is, the less similar the strings are. The Levenshtein distance can be used to compute the similarity value between two strings in the following way:

\[
similarity(x, y) = 1 - \frac{\text{levenshtein}(x, y)}{\max(\text{length}(x), \text{length}(y))}
\] (2.2)

In this example, the similarity between the strings ‘filipa’ and ‘philippa’ is around 0.625.
2) Jaro–Winkler distance

The Jaro-Winkler distance [37] was proposed to deal with cases where two strings that have a common prefix result in a low value with the Jaro measure [24] [25]. In order to better understand the Jaro-Winkler distance we will first start by explaining the Jaro measure.

The Jaro measure is computed with the following steps:

1. Find the number of common characters $c$ that are present in both strings. However, the distance between the characters in both strings must be below a certain value. For example, let us assume that we have a character $x_i$ in the string $x$ at position $i$ and we have the same character $y_j$ in the string $y$ at position $j$. We can only count these two characters as common if $|i - j| \leq \frac{\min(|x|,|y|)}{2}$. In Figure 2.2 we have an example with two strings. Since 'telma' is shorter than 'thelma', then the $\frac{\min(|x|,|y|)}{2}$ is $\frac{5}{2} = 2.5$. In this case the common characters are ['t', 'e', 'l', 'm', 'a'], which means $c = 5$.

![Figure 2.2: Finding the common characters between two strings with the Jaro measure.](image)

2. The next step is to compare the $i$th common character of one string with the $i$th common character of the other string. If the character is not the same then it is a transposition. The goal of this step is to count the number of transpositions $t$. Using the same example from step 1, since the common characters of both strings have the same sequence, then $t = 0$.

3. After having $c$ and $t$ defined we can now compute the Jaro measure with the following formula:

$$jaro(x, y) = \frac{1}{3} \times \left[ \frac{c}{|x|} + \frac{c}{|y|} + \frac{(c - t)}{c} \right]$$

(2.3)

In this example we have $\frac{1}{3} \times \left[ \frac{5}{5} + \frac{5}{5} + \frac{(5-0)}{5} \right] = 0.94(4)$. As we can see, the similarity value is high, which means that these strings are most likely to be the same. Let us assume that there is some typing error and instead of 'thelma' we have 'thelam'. In this case, the common character sequence of 'telma' is 'telma' and the common character sequence of 'thelam' is 'telam', which means $c = 5$ and $t = 2$ because we need to change 'm' to 'a' and 'a' to 'm'. Due to this mistyping, the Jaro measure is around 0.88.

The Jaro-Winkler measure tries to overcome these mistyping situations by taking into account the $PL$ which is the length of the longest common prefix between the two strings. In the example of 'telma' and 'thelam', the longest common prefix is 'tel', so $PL = 3$. To this prefix we also assign a weight $PW$, which means the relevance of the prefix in determining whether two strings are alike or not. For this example we will assume that $PW = 0.1$. The Jaro-Winkler distance is given by the following formula:
\[ jaro - winkler(x, y) = (1 - PL \times PW) \times jaro(x, y) + PL \times PW \] (2.4)

In this example, with the strings ‘telma’ and ‘thelam’, the Jaro-Winkler distance value is around 0.9.

**Token-based string matching algorithms**

The token-based string matching algorithms calculate the similarity between two strings based on sets of tokens. A token is a value extracted from a given string. One way to extract these values is to consider the words in a string delimited by a space. For example, for the string ‘telma filipa fernandes’ we have 3 tokens: ‘telma’, ‘filipa’ and ‘fernandes’. Another type of token is a \(q\)-gram. A \(q\)-gram is a substring of length \(q\) that is extracted from the string. For example, the \(q\)-grams of size 2 (i.e., bigrams) of the string ‘telma’ are [‘#t’, ‘te’, ‘el’, ‘lm’, ‘ma’, ‘a#’]. The similarity values are then computed based on the frequency of the tokens in both strings.

We explain two algorithms that apply the token-based approach in order to determine if two strings match.

1) **Jaccard similarity coefficient**

The Jaccard similarity coefficient \([22][23]\) starts by generating a set of tokens for both strings. For example, for string \(x = \text{telma}\) and \(y = \text{thelma}\) and using bigrams, the tokens are \(B_x = \{#t, te, el, lm, ma, a\#\}\) and \(B_y = \{#t, th, he, el, lm, ma, a\#\}\). The Jaccard similarity value is calculated with the following formula:

\[ jaccard(x, y) = \frac{B_x \cap B_y}{B_x \cup B_y} \] (2.5)

In this example the similarity value is \(\frac{5}{8} = 0.625\).

2) **Term Frequency/Inverse Document Frequency**

The Term Frequency/Inverse Document Frequency (TFIDF) approach \([33][34][32]\) is based on the frequency of the common tokens in both strings. In other words, this algorithm considers two strings as similar if they share distinguishing terms. For example, consider strings \(x = \text{Telma Fernandes, MSc student}\), \(y = \text{Filipa Fernandes, MSc student}\) and \(z = \text{Telma F}\). If we applied, for example, the Jaccard similarity coefficient to these strings, it would produce an higher similarity value for strings \(x\) and \(y\) than for strings \(x\) and \(z\). In this case, the Jaccard similarity would not produce an accurate result, because records \(x\) and \(z\) belong to the same entity and records \(x\) and \(y\) refer to different entities. The TFIDF can acknowledge that ‘Fernandes’, ‘MSc’ and ‘student’ are common words and that ‘Telma’ is a distinguish term. With this knowledge, the TFIDF approach can correctly identify \(x\) and \(z\) as a match.

As an example we will consider strings \(x = \text{telma}\) and \(y = \text{thelma}\). In order to compute de TFIDF we need to compute the Term Frequency (TF) and the Inverse Document Frequency (IDF) of both strings. The TF corresponds to the total number of strings that contain that token. For example, for the two strings ‘telma’ and ‘thelma’, the token ‘el’ is present in both strings, so its value is 2. The IDF is the
number of times the token occurs in the string that generated it. For example, the token 'el' appears only once in string 'telma', so its IDF is 1.

The TFIDF algorithm consists in the following steps:

1. For each string we compute its bag of tokens. For this example we will consider that the tokens are the letters from each string. For string $x$ we have $B_x = \{t, e, l, m, a\}$ and for string $y$ we have $B_y = \{t, h, e, l, m, a\}$.

2. The next step is to compute the TF of each bag. For each unique character we verify its occurrence in each bag. As we can see in Figure 2.3(a) we have a matrix where each column corresponds to each character and each row to each bag. In each field we have the occurrence of that character in that bag.

3. In order to compute the IDF, we divide each TF with the square-root of the size of the bag. In figure 2.3(b) we have the IDF of each character in each bag for the example we are following.

4. After all these steps we can compute the similarity value that is given by the sum of the multiplication of both IDF for each character. In this example the similarity value is given by $0.447 \times 0.408 + 0.447 \times 0.408 + 0.447 \times 0.408 + 0.447 \times 0.408 + 0.447 \times 0.408 \approx 0.912$.

Phonetic string matching algorithms

These string matching algorithms are very different from the previous ones. Until now we described string matching algorithms that compute the similarity value between two strings based on how those strings are written. Phonetic similarity measures compute the similarity value based on how the two strings sound.

We are going to describe one phonetic similarity measure and its later improvement.

1) Soundex

The approach of the Soundex algorithm [36] consists in converting each string into a code of four characters that represents the sound made by that string. In this example we will consider the strings $x = \text{Filipa}$ and $y = \text{Filippa}$.

The Soundex algorithm consists in the following steps:

1. The first letter of each string is selected as the first character of the code. For both strings we have 'F' as the first letter of the code.
Table 2.1: Conversion table used by the Soundex algorithm.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>b, f, p, v</td>
<td>1</td>
</tr>
<tr>
<td>c, g, j, k, q, s, x, z</td>
<td>2</td>
</tr>
<tr>
<td>d, t</td>
<td>3</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>m, n</td>
<td>5</td>
</tr>
<tr>
<td>r</td>
<td>6</td>
</tr>
</tbody>
</table>

2. The second step is to remove all occurrences of the characters ‘w’ and ‘h’. Since in both strings we do not have any occurrences of those characters, the strings stay the same. Next we exchange each consonant (except for the first one) with a digit as in Table 2.1. For string \( x \) we have ‘Fi4i1a’ and for string \( y \) we have ‘Fi4i11a’.

3. The third step is to reduce each sequence of identical characters to one. So ‘Fi4i11a’ stays ‘Fi4i1a’.

4. The next step is to remove all non-numeric characters (except for the first one). For both strings we have ‘F41’. The resulting code consists in the first four letters of the code produced so far. Since there are not enough digits we add zeros until the code has four digits. In conclusion, the code for both string is ‘F410’ which means that both strings sound the same. The Soundex algorithm, based on the produced code, states that both string \( x \) and \( y \) are similar.

2) Metaphone

Although Soundex match most of the strings that sound the same there are still some cases that this phonetic algorithm cannot cover. For example, ‘Filipa’ and ‘Philippa’ produce the codes ‘F410’ and ‘P410’, respectively. This means that both strings do not match, when in fact the name Filipa is the Portuguese version of the name Philippa.

The Metaphone algorithm [30] is an improvement of the Soundex algorithm. This algorithm has several rules to be applied to each string in order to produce their corresponding code. In order to simplify this explanation we will only mention the rules that are applied to strings \( x = \text{Filipa} \) and \( y = \text{Philippa} \).

1. The first rule to be applied is to drop the duplicate adjacent letters (except for the letter C). This rule is applied more specifically to string \( y \) which becomes ‘Filipa’.

2. The second rule is applied once again to string \( y \), which consists in transforming ‘PH’ into ‘F’. So ‘Philipa’ transforms into ‘Filipa’.

3. The last rule to be applied is to drop all the vowels (except if it is in the beginning of the string).

By applying this rule, the code that is produced for both strings is ‘Flp’. In conclusion, \( x \) and \( y \) are similar.

2.1.2 Rule-based Matching

The Rule-based Matching approach consists in applying a set of matching rules, defined by the user, to each pair of records in order to determine if they match or not. The rules are typically similarity tests
applied to one or more attributes of the two records. The results of these tests are then combined to produce a final similarity value for the pair of records.

An example of a rule applied to a pair of records \(x\) and \(y\) is 
\[
\text{sim}(x, y) = (\text{sim}_{\text{surname}}(x, y) \geq 0.9) \land \text{sim}_{\text{SSN}}(x, y) = 1.0) \Rightarrow [x, y] \rightarrow \text{MATCH},
\]
where \(\text{sim}_{\text{surname}}(x, y)\) calculates the similarity value of the record’s attribute surname and \(\text{sim}_{\text{SSN}}(x, y)\) returns the similarity value of the attribute Social Security Number (SSN). If the conjunction of the two similarity tests is true, then the pair of records \(x\) and \(y\) is considered a match.

Preferably, matching rules must have a high accuracy and a high coverage. A rule has high accuracy when it classifies most of the pairs correctly. A high coverage is when a rule covers a significant amount of all candidate pairs. In general, the more specific a rule is the more accurate it is, however its coverage is lower. This two characteristics define the quality of the rules.

There are several approaches to build rules. We are going to briefly describe two of them.

**Traditional approach**

Matching rules are manually generated by the user and require a previous knowledge of the data. The creation of such rules is a very labor-intensive task, specially if we are dealing with a large amount of records. After generating a set of possible rules, the user has to test them manually in order to evaluate their quality and, if needed, refine them. This is an iterative process that can be very tedious for the user.

**Learning rules from training data**

Alternatively, the generation of rules can be performed by learning them from a training data that has examples of pairs of records with their true match status [18]. The rules are generated based on this training data. The learning process consists of the following steps:

1. The training data is constituted by the dataset and a set of rules that cover the pairs of records that are true matches and a set of rules that cover the ones that do not match. An example of a rule generated from the training data is 
\[
\text{sim}_{\text{name}}(x, y) = 1.0) \Rightarrow [x, y] \rightarrow \text{MATCH},
\]
where \(\text{sim}_{\text{name}}\) calculates the similarity (e.g, with a string matching algorithm) of the value in the attribute name. This rule was created because each pair of records in the training data that had the same value in the attribute name corresponds to a match. At the end of this learning step we obtain a set of rules that were generated from the training data.

2. From all the rules that were generated from the training data, we choose the one that is the best candidate rule. This selection takes into account the accuracy and coverage of each rule [18]. The method selects the rule which has the best trade-off between accuracy and coverage. After choosing the best candidate rule, we expand it with new candidate rules. In other words, we add more conditions to the chosen rule. Assuming that the rule 
\[
\text{sim}_{\text{name}}(x, y) = 1.0) \Rightarrow [x, y] \rightarrow \text{MATCH}
\]
was chosen as the best candidate rule, then this rule is expanded with more conditions (i.e., a new set of candidate rules). One example of an expansion is 
\[
\text{sim}_{\text{name}}(x, y) = 1.0) \land \text{sim}_{\text{SSN}}(x, y) \geq 0.8) \Rightarrow [x, y] \rightarrow \text{MATCH}.
\]
This rule was generated because all pairs of records that share the same value in the attribute name have at least 80% of similarity in the attribute SSN.
At the end of this step, the method generates a set of expanded rules that were learned from the training data along with the rule that was chosen as the best candidate rule.

3. From all the expanded rules the method chooses, once again, the best candidate and repeat the process until a stopping criteria is met. After the end of these phases, all the candidate record pairs that are covered by the generated rule are removed from the training data. If there are record pairs that are still left in the training data, then we create a new rule, repeating all the process from the beginning, but only with those remaining record pairs.

2.2 Scaling Up Rule-based Matching

This Section describes the most relevant techniques that were proposed to optimize the approximate duplicate detection. The main goal of these algorithms is to improve the efficiency of the approximate duplicate detection process, in order to avoid a performance bottleneck when analyzing a large amount of data.

In this Section we explain different techniques for scaling up the Rule-based matching. Section 2.2.1 explains the Traditional Blocking, an indexing technique used in data matching which assigns all similar records into the same block. In Section 2.2.2 we detail the Sorted-Neighborhood Join method, as well as some variants of the algorithm that were further proposed. Section 2.2.3 explains the Q-gram Based Indexing which assigns all records that have the same variation of their key into the same cluster. Section 2.2.4 describes the Suffix Array Based Indexing, which is very similar to the Q-gram Based Indexing, but uses the record's key suffix instead of using variations of the record’s key. Section 2.2.5 explains the Canopy Clustering which is an algorithm that uses computationally cheap similarity measures to assign records to overlapping clusters. Finally, in Section 2.2.6 we will summarize all the algorithms that were explained in this Section.

2.2.1 Traditional Blocking

The Traditional Blocking technique [14] is one of the oldest optimizations for duplicate detection algorithms and it is based on a simple indexing mechanism. The main purpose of the use of indexing in data matching is to reduce the amount of records that are compared by immediately excluding those who are not true matches.

The first step of this approach is to create, for each record, a key that is based on the values of one or more attributes. In Figure 2.4(a) we have an example of some records and their corresponding key values. After this step, all records that have the same key value are inserted in the same block. A block, in this case, is a group of records that are possible approximate duplicates.

As we can see in Figure 2.4(b) only the records that have the key ‘telma’ are in that block. The records that have the key ‘thelma’ are in the other block, as shown in Figure 2.4(c).

The last step is to compare the records that are in the same block. More specifically, in the first block \( r_1 \) is compared with \( r_3 \) and in the second block \( r_2 \) is compared with \( r_4 \). In conclusion, only the records
Figure 2.4: Example of applying the Traditional Blocking to a set of records.

within the same block are compared among each other.

2.2.2 Sorted-Neighborhood Join

The Sorted Neighborhood Join algorithm [19] was introduced to match similar records in a single table or belonging to several tables. This technique sorts the records according to a key, instead of generating blocks. This key is also based on the values of one or more attributes. If the records belong to multiple tables, all records are first concatenated into a single table and then sorted according to the chosen key. The Sorted-Neighborhood Join (SNJ) algorithm has the following three phases:

1. **Create Keys**: For each record in the table, the algorithm extracts one or more attributes (or portions of them) to compose a key (as previously defined by the user). In Figure 2.5(a) we have an example with a set of records and their corresponding key values.

2. **Sort Data**: The records are ordered according to the key computed in the previous step. The purpose of sorting is to guarantee that all records that are possible matches stay close to each other. In Figure 2.5(b) we have the records sorted by their key. The effectiveness of SNJ depends on the quality of the chosen key, because if similar records have different key values they will be far from each other after this sorting phase and it may not be possible to compare them.

3. **Merge**: In this step, we sequentially move a window with fixed size (greater than 1 and less than the total number of records) over the sorted records. In Figure 2.5(c) we have a sliding window of size 3 that starts at the beginning of the sorted table.

In the first iteration, the records \( r_2, r_4 \) and \( r_6 \) will compose the first candidate record pairs. The resulting candidate record pairs from the window are \( \{ (r_2, r_4), (r_2, r_6), (r_4, r_6) \} \). After this, the window slides down one position to cover \( r_4, r_6 \) and \( r_3 \). These records will compose the new candidate record pairs.

The sliding window will go through all the records and generate all the candidate records pairs until it reaches the end of the table.

The size of the window has a significant impact in the merge phase, because the larger the window, the higher the probability of comparing records that are potentially approximate duplicates. However, the
computational complexity increases proportionally to the window size, because the number of comparisons performed with this method is $O(nw)$, where $n$ is the total number of records and $w$ is the window size.

**Clustering Method**

The Clustering Method approach [19] was proposed as an alternative to the traditional SNJ. It is based on partitioning the initial data records into independent clusters.

When dealing with multiple tables, all records are concatenated into one single table. The clustering method can be summarized in the following two phases:

1. **Cluster Data:** The algorithm assigns each record to its corresponding cluster based on an extracted key from each record. This key, like in the traditional SNJ, is based on one or more attributes. At the end of this phase, all possible matching records will belong to the same cluster.

2. **Sorted-Neighborhood Method:** The algorithm applies the SNJ independently to each cluster created in the previous step. Since it has already computed a key for each record, the first phase of the SNJ (i.e., creation of keys) can be avoided.

**Multi-pass Approach**

The effectiveness of the SNJ highly depends on the key used to sort the data. However, real-world data is dirty and using one key may not be enough to detect all matching records. The attribute (or part of it) that appears first in the key has a higher discriminating power, and sometimes that attribute may not be totally correct. For example, consider two records referring to the same entity, whose key is composed by the Social Security Number (SSN). One of the records has the SSN 192345678 and the other one has 912345678. After ordering all the records in the table by the key value, it is almost impossible that these two records fall within the same window. In order to overcome this situation, a solution is to execute several runs of the SNJ independently using a different key on each run. This strategy is known as the Multi-pass Approach [19].
Each iteration of the multi-pass approach produces a set of pairs of candidate duplicate records. Consider the following situation where we have three pairs: \((r_1, r_2), (r_1, r_3)\) and \((r_2, r_3)\). The pairs \((r_1, r_2)\) and \((r_1, r_3)\) were considered a match, but pair \((r_2, r_3)\) was considered as a non-match. If both \(r_2\) and \(r_3\) are considered the same entity as \(r_1\), then they should also be considered the same entity. Because of these situations, the algorithm applies a transitive closure. Since both \(r_2\) and \(r_3\) matched with \(r_1\), then by transitive closure, the pair \((r_2, r_3)\) will be also a match.

In this approach, the use of the transitive closure enables to discover more candidate record pairs. For example, consider that the first iteration of the multi-pass with key \(k_1\) results in record \(r_1\) being a duplicate of \(r_2\), and the second run of the SNJ with key \(k_2\) produces \(r_2\) and \(r_3\) as duplicates. Since \(r_1\) is a duplicate of \(r_2\) and \(r_2\) is a duplicate of \(r_3\) then, by transitive closure, \(r_1\) is a duplicate of \(r_3\). This approach enables to discover more duplicates even if they do not match in the same run of the SNJ. The final result of the Multi-pass Approach is a union of all the pairs discovered along the various iterations of the multi-pass (plus the ones inferred by transitive closure).

**Incremental Merge/Purge**

All variants of the SNJ algorithm start by first concatenating all records from the various input tables into a single one. After applying the SNJ algorithm, the data in the table is considered to be cleaned. If new records are inserted into the table, then they should be concatenated to the existing cleaned table before re-applying the SNJ algorithm. If we are dealing with large amounts of data, concatenating all records again will not be the most efficient approach, since most of the records are already cleaned. One way to overcome this situation is to apply the Incremental Merge/Purge algorithm. This algorithm executes in several iterations.

The first iteration of this algorithm consists in applying the multi-pass method to the input relation. Then, it creates clusters of similar records. For each cluster, the algorithm extracts a set of records to represent the records in that cluster. This set of records is called prime-representatives.

In the remaining iterations of the Incremental Merge/Purge algorithm, the input relation is concatenated with the prime-representatives. After applying the multi-pass method, each record is added to the existing clusters or to a newly created cluster. Once more, the prime-representatives are extracted from each cluster. This whole process repeats every time new records are inserted.

The accuracy of the Incremental Merge/Purge algorithm depends on the correct selection of the prime-representatives. There are some strategies for selecting the prime-representatives. One example is to select a random sample of records from each cluster and another example is to choose the last record inserted in each cluster.

**Inverted Index Based Approach**

In the SNJ algorithm, the records are sorted according to the key value and then compared among each other inside a fixed-size sliding window. One problem with this approach can occur if we have more records with the same key value than the fixed size of the sliding window. If this happens, then not all
records with the same key value are going to be compared within the same window. A solution proposed for this problem is to use an inverted index table. An inverted index table is a data structure that maps a key value to its corresponding records. In this case, the inverted index keys are all the unique key values generated from all records. For each inverted index key we will have an inverted index list of a set of identifiers of all the records that generated that key value.

The Inverted Index Based Approach \[4\] starts to assign an unique identifier to each record. After the identification step, we compute a key for each record based on one or more attributes. Just like in the traditional SNJ algorithm, the records are sorted according to that key. After the sorting step, an inverted index table is created in which each unique key value is associated to a list with all the identifiers of the records with that same key value. The sliding window passes through the inverted index table and, just like in the SNJ, compares all the records, that are inside the sliding window, among each other.

This approach enables to compare all records with the same key value, no matter the window size, because each unique key only appears once in the inverted index table. In the traditional SNJ, the same key value appeared as many times as the number of records that had that same key value.

**Adaptive Sorted Neighborhood Approach**

One of the major problems of the SNJ algorithm is due to the fixed window size. If the window size is too small it may not cover all true matches if similar records are not close enough to each other. If the window is too large then there will be a lot of records that are not true matches that will be compared.

The Adaptive Sorted Neighborhood Approach \[38\] is very similar to the traditional SNJ, with the slight difference that it creates various windows with different sizes.

This approach starts the same way as the traditional SNJ. Then, after the sorting phase, a string similarity function is applied to the key values of the records adjacent to each other in the sorted table. The goal of this comparison is to find out if the similarity between adjacent records is below a certain minimum threshold \(t\). If their similarity is below \(t\), then the algorithm considers that those records are significantly different. If two adjacent records satisfy this condition, then this pair of records is a boundary pair \[38\]. The purpose of the boundary pairs is to define the end of a window and the start of a new one. The size of each window varies and it depends on the amount of similar records that are between the range of each boundary pair.

The main purpose of this approach is to guarantee that the records that stay outside a window are significantly different from the ones within that window. This way, the probability of covering all true matches becomes higher than having a fixed size window.

**2.2.3 Q-gram Based Indexing**

In the Traditional Blocking technique there is a point of failure. For example, supposing the attribute(s) on which the key is going to be created has errors, then the record may not fall under the correct block. The Q-gram Based Indexing \[3\] aims at overcoming this limitation by inserting records that have a similar key into the same block.
This approach is composed in the following three phases:

1. **Create Q-gram:** Just like in the *traditional blocking*, each record has a key based on one or more attributes. In the first phase, this method converts each key value into a list of q-grams. A q-gram is a substring of length q. The value of q is chosen by the user, but for this explanation, q = 2 (bigrams). A key with c characters will have \( k = c - q + 1 \) q-grams [6]. The approach that is used to create these q-grams consists in using a sliding window which extracts q characters at any position from the start to the end of the key. For example, the bigram list created from the key 'telma' is ['te', 'el', 'lm', 'ma'].

2. **Create Key Variations:** The algorithm generates all possible combinations with the q-grams created in the previous phase. The created q-gram variations must have a minimum length \( l \). This minimum length avoids having key variations too short in which almost all records could generate that key variation. To compute the minimum length \( l \), the user must define a minimum threshold \( t \). Then, \( l = \max(1, \lfloor k \times t \rfloor) \). Assuming \( t = 0.75 \) and the same example from above, the minimum length of the q-gram sublists is \( l = 3 \).

In Figure 2.6, we have an example of applying the q-gram based indexing to two records, where \( q = 2 \) and \( t = 0.75 \). In the column 'Record ID' we have the record identifier and in the column 'Key value' we have the key value of each record. In the column 'Bigram sublists' we can observe the possible combinations that were generated with the bigram list created in the first phase.

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Key value</th>
<th>Bigram sublists</th>
<th>Index key values</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>telma</td>
<td>[te,el,lm,ma], [te,el,lm], [te,el,ma], [te,lm,ma], [el,lm,ma]</td>
<td>teellma, teelm, teelma, telma, elmma</td>
</tr>
<tr>
<td>r₂</td>
<td>thelma</td>
<td>[th,he,el,lm,ma], [th,he,el,lm,ma], [th,he,el,ma], [th,he,lm,ma], [he,el,lm,ma], [he,el,ma], [he,lm,ma], [he,lm,ma], [el,lm,ma]</td>
<td>thheellma, thheellm, thhelimma, thheimma, thheel, thhelm, thheelma, thelma, thelma, helimma, helma, elmma</td>
</tr>
</tbody>
</table>

Figure 2.6: Example of applying the q-gram based indexing to two records.

3. **Create Index Keys:** All bigram sublists are converted into strings by concatenating the q-grams. These strings are the candidate index keys of the corresponding record. If more than one record generates the same index key value then that candidate key becomes an index key for those records. In the column 'Index key values' of the table represented in Figure 2.6, we can observe the generated index key values for those two records. Since key 'ellma' (highlighted in bold) was generated both for \( r₁ \) and \( r₂ \), then those records are inserted into the block that corresponds to key 'ellma'.

At the end of the three phases, each record is assigned to the various blocks generated by the
created index keys. All the records that are inserted into the same block are further compared in order to determine if they are matches.

### 2.2.4 Suffix Array Based Indexing

The Suffix Array Based Indexing [1] technique is very similar to the Q-gram Based Indexing technique. It also creates variations of the key, but it uses the key's suffix instead of using q-grams.

In this technique, the user must set a minimum length \( l_m \) for each suffix. A key with \( c \) characters generates \( (c - l_m + 1) \) suffixes. For example, the key ‘fernandes’ with \( l_m = 5 \), generates ‘fernandes’, ‘ernandes’, ‘rnandes’, ‘nandes’ and ‘andes’. In Figure 2.7(a) we have an example of the suffixes generated by each record’s key value.

Each unique suffix is an index key that is inserted into an inverted index table. This table contains for each index key value, a list containing all the records that originated that index key value. In Figure 2.7(b) we have an example of the created inverted index table for the records in Figure 2.7(a) using \( l_m = 4 \) and \( b_M = 2 \).

![Figure 2.7: Example of applying the suffix-array based indexing to a set of records.](image)

This approach may result in having a lot of records associated to one index key value, so the user must also define a maximum block size \( b_M \) which limits the number of records that a block may contain. If an entry in the inverted index table contains more than \( b_M \) records, it is removed. For example, let us assume that \( b_M = 2 \). In Figure 2.7(b), the suffix ‘rina’ has more records associated than the value defined in \( b_M \), so that entry is removed from the inverted index table.

Each suffix corresponds to a block that contains the records associated in the inverted index table. The records that belong to the same block are compared among each other.
Robust Suffix Array Based Indexing

The Robust Suffix Array Based Indexing [9] is an improvement of the Suffix Array Based Indexing. It merges the index entries whose index key values are similar. To do this, the algorithm applies an approximate string similarity measure to all pairs of index key values. If the similarity value between the suffix values is greater than or equal to a given threshold $t$, then the corresponding record identifiers are merged, thus forming a larger list for those two index key values.

Considering the example of the inverted index table in Figure 2.7(b), assume that $t = 0.85$ and that the result of applying the approximate string similarity measure to the pairs 'cathernina' and 'katherina' is 0.89. Since this similarity is greater than the defined threshold, then their record identifier lists $[r_1]$ and $[r_2]$ are merged, resulting in list $[r_1, r_2]$. Instead of having two blocks for each index key value, we only have one block for both index key values. The records inside that block are then compared among each other.

2.2.5 Canopy Clustering

The Canopy Clustering method [8] [27] uses a computationally cheap similarity measure to group the records into overlapping clusters, also called canopies. Inside each canopy, the records are compared among each other with the use of another similarity measure. This similarity measure, normally, is more computationally expensive than the one used to group the records.

This method starts by creating an inverted index table with tokens. A token can be a word, a character or a $q$-gram. For each key value we can generate one or more tokens. There are several ways to form these tokens, but for this explanation we are going to consider that the tokens are bigrams of the record’s key.

This Canopy Clustering method starts by, for each record, generating the tokens from its key. Since we are considering the tokens as bigrams, this step is performed the same way as the creation of $q$-grams in Section 2.2.3. Each unique token generated is inserted into an inverted index table, along with a list of the corresponding records that contain that token.

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Key value</th>
<th>Sorted token list</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>telma</td>
<td>$[el, 1], (lm, 1), (ma, 1), (te, 1)]$</td>
</tr>
<tr>
<td>$r_2$</td>
<td>thelma</td>
<td>$[el, 1], (be, 1), (lm, 1), (ma, 1), (th, 1)]$</td>
</tr>
</tbody>
</table>

(a)

![Figure 2.8: Example of applying the Canopy Clustering to two records.](image)

Figure 2.8 illustrates a simple example of applying the Canopy Clustering to two记录, where the tokens are bi-grams extracted from the record's key. In Figure 2.8(a) we have the records, their corresponding key value and the sorted bigram list extracted from the key value. Each token is paired with a numerical value which is the Document Frequency (DF). The DF is the number of times the token
occurs in the key that generated it. For example, the token ‘el’ appears only once in the key value ‘telma’, so its DF is 1.

Each unique token is inserted into the inverted index table, along with the list of the corresponding records whose keys contain that token. As represented in Figure 2.8(b), each token has a numerical value associated to its value. That numerical value is the Term Frequency (TF). The TF corresponds to the total number of records that contain that token. For example, token ‘el’ is present both in the key ‘telma’ and ‘thelma’, so its value is 2.

After the creation of the inverted index table, the canopies are created. This step can be performed in two different ways: based on thresholds or based on nearest-neighbors.

**Threshold Based Approach**

In this approach, the user must define two thresholds, a loose threshold \( t_l \) and a tight threshold \( t_t \). It starts by inserting all record identifiers into a set \( P \). Then, one record identifier \( r_c \) is randomly extracted from \( P \). For every remaining records in \( P \), the method selects the ones which have, at least, one token in common with all the tokens in \( r_c \). Note that this step is more efficient when using the created inverted index table, because it shows, for each token, all the records whose key generated that token.

For all the records \( r_x \) in \( P \) that have, at least, one token in common with all the tokens in \( r_c \), the method applies a similarity function to compare those records with \( r_c \). If \( r_x \) has a similarity value with \( r_c \) above \( t_t \), then \( r_x \) is inserted into the same canopy as \( r_c \). If \( r_x \) has a similarity value with \( r_c \) above \( t_t \), then \( r_x \) is removed from \( P \). After having identified all records that belong to \( r_c \)’s canopy, \( r_c \) is removed from \( P \). The whole process is repeated as long as there are record identifiers in \( P \).

At the end of this process, all canopies contain the records that are similar among each other. Note that there may be some records that appear in more than one canopy.

**Nearest Neighborhood Based Approach**

In this alternative, the user must define two nearest neighbor parameters, \( n_t \) and \( n_l \) (where \( n_t \) cannot be greater than \( n_l \)). The parameter \( n_t \) is the number of records that are removed from \( P \) at each iteration. The parameter \( n_l \) corresponds to the number of record identifiers that are assigned to the same cluster as the record chosen for centroid. The centroid of each cluster is a random record \( r_c \) extracted from \( P \) (which initially has all records).

For every record \( r_x \) left in \( P \), the method applies a similarity function to compare all records with \( r_c \), as long as they have at least one token in common. Just like in the threshold based approach, we can use the inverted index table to optimize this process. The \( n_t \) records that are closest to \( r_c \) are inserted into the same cluster as \( r_c \), and the \( n_l \) closest records to \( r_c \) are removed from \( P \). At the end of creating the \( r_c \)’s canopy, \( r_x \) is also removed from \( P \). This algorithm continues iteratively until \( P \) is empty.

The Nearest Neighborhood Based Approach enables to know exactly how much records are in each canopy. However, if the \( n_t \) value is too small, some true matches may not fall under the same cluster because it can be full.
2.2.6 Discussion

In this Section we present a table that summarizes all the algorithms to scale up the Rule-based matching that were described in this Section.

In Table 2.2 we have the data matching algorithms described in this Section. The column 'User input' shows what the user must define manually in the algorithm. The column 'Uses key' tells us in which way the created key for each record is used during the execution of the algorithm. The column 'Groups records' confirms whether the records are grouped (e.g., in a cluster, block, list, etc) in order to be compared among each other.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>User input</th>
<th>Uses key</th>
<th>Groups records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional blocking</td>
<td>Key formation</td>
<td>Group records</td>
<td>Yes</td>
</tr>
<tr>
<td>Sorted Neighborhood Join</td>
<td>Key formation</td>
<td>Sort records</td>
<td>No</td>
</tr>
<tr>
<td>Clustering method</td>
<td>Key formation</td>
<td>Sort records</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-pass approach</td>
<td>Keys formation</td>
<td>Sort records</td>
<td>No</td>
</tr>
<tr>
<td>Incremental merge/purge</td>
<td>Keys formation</td>
<td>Sort records</td>
<td>Yes</td>
</tr>
<tr>
<td>Inverted index based approach</td>
<td>Key formation</td>
<td>Form indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Adaptive approach</td>
<td>Key formation</td>
<td>Sort indexes</td>
<td>No</td>
</tr>
<tr>
<td>Q-gram based indexing</td>
<td>Key formation</td>
<td>Form indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suffix array based indexing</td>
<td>Minimum index length</td>
<td>Form indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Canopy clustering</td>
<td>Key formation</td>
<td>Form tokens</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Token formation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Threshold or neighbor params</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Classification of the scale-up algorithms explained in this Chapter.

As we can see in Table 2.2 the user needs to specify a key for all the scale-up algorithms that we discussed in this Chapter. The quality of the chosen key is one of the factors that determine the effectiveness of these algorithms. The keys are mostly used to sort/group records or form indexes/tokens.

In all the variations of the SNJ algorithms, the window size is also very important to produce accurate results. The size of the window cannot be neither too small nor too large. If we have a small window, then we can miss some pairs of records that are true matches because they will never be under the same window. If the size of the window is too large, then the complexity of the algorithm will be bigger because there will be a lot of comparisons between records that do not refer to the same entity.

In conclusion, if the user does not choose the most suitable parameters, for the dataset to be analyzed, these algorithms may not produce the best results, both in terms of efficiency as effectiveness.
Chapter 3

CLEENEX

In this Chapter we present CLEENEX, a prototype for relational data cleaning based on AJAX [16]. AJAX is a data cleaning framework that enables the specification of data cleaning programs using a language that extends SQL. AJAX was extended [17] in order to support Quality Constraints (QCs) and Manual Data Repairs (MDRs). CLEENEX [10] is the result of the implementation of QCs and MDRs in AJAX.

In Section 3.1, we define how a data cleaning program is developed in CLEENEX. In Section 3.2 we explain the various steps that are part of the core operation of CLEENEX. In Section 3.3 we detail the environment through which the user can interact with CLEENEX either to resolve data inconsistencies and errors that were not automatically handled or to refine the data cleaning program. In Section 3.4 we explain each component of the CLEENEX architecture and detail the important operational units that compose the core of the CLEENEX system.

3.1 Specification of data cleaning programs

The CLEENEX framework provides a clear separation of the logical and physical level. At the logical level, the user specifies the sequence of data transformations. At the physical level specific algorithms can be selected to implement the data transformations.

3.1.1 Logical level

A data cleaning program in CLEENEX is modeled as a Directed Acyclic Graph (DAG) that corresponds to the workflow of data transformations to be applied to a dataset. Each of these transformations is defined through a logical operator. CLEENEX supports the following five logical operators [15]:

- **View**: An arbitrary SQL query.

- **Map**: Establishes a one-to-many mapping between an input record and the corresponding output records. In other words, each record from the input relation can produce zero or more records into the output relation.
• **Cluster:** For a given input relation, it groups its records, according to a given clustering algorithm (e.g., *transitive closure*).

• **Merge:** Groups the records of an input relation, by a given criteria, and chooses a representative for each group. This transformation is mostly used to eliminate approximate duplicates that are found by the *matching* transformation.

• **Match:** Applies an approximate join to two input relations. This transformation is typically used to detect approximate duplicate records.

CLEENEX also supports the use of external functions (implemented in Java) to be invoked within operators. All the functions are defined by the user and are stored in a library of external functions. The external functions can be, for example, string matching algorithms that can be used by the match operator.

### 3.1.2 Physical level

At the physical level, certain decisions can be made in order to optimize the execution of a data cleaning program. More specifically, an efficient approach can be chosen to implement a logical operator. For example, in the case of the *View* operator (which is a simple SQL query), executing it directly inside the Relational Database Management System (RDBMS) can bring some optimization to the data cleaning program. The reason is because most RDBMSs already provide mechanisms that determine the most efficient way to execute a given query.

In particular, for the *Match* operator, its naive implementation consists in the computation of a Cartesian product between the two input tables. For each candidate record pair it applies a similarity function in order to determine whether the two records match. The *Match* operator is one of the most expensive operators, specially if the two input tables contain a large amount of records. Therefore, it is fundamental that CLEENEX supports physical algorithms for the *Match* operator that are more efficient than the computation of the Cartesian product.

### 3.2 Core operation of CLEENEX

The core operation of CLEENEX consists in the following two steps:

1. **Compilation:** The data cleaning program written by the user is analyzed (i.e., parsed) in order to discover any syntactic errors. Each data transformation of the data cleaning program has a syntactical specification that consists in the following clauses:

   (a) **CREATE:** This clause specifies the type of data transformation and the name of the output relation of that transformation.

   (b) **FROM:** This clause states the input relation(s) of the transformation.
(c) **LET:** This clause is optional. The use of this clause enables the user to specify a given function (e.g., a string matching function) to be applied to the input records.

(d) **WHERE:** This clause is used to filter out the input records that do not satisfy a given condition.

(e) **SELECT:** This clause is used to define the schema of the output relation after applying the transformation.

(f) **BY:** This clause is used by the *Cluster* operator to specify the clustering algorithm to be used.

Listing 3.1 illustrates an example of a data transformation that complies the syntactical rules defined by CLEENEX. In the FROM clause we have defined the two input relations: *Authors* and *TeamAuthors*. The relation *Authors* contains all the names of the authors and relation *TeamAuthors* contains all the names of the authors of a given team. This transformation applied a *Cartesian Product* to the records from both tables in order to produce the candidate record pairs. The LET clause invokes the function *similarAuthors* to be applied to each pair of records to compute their similarity value. If the resulting value of that string matching functions is at least 0.93 then those records will be considered as matches. The pairs that are considered as matches are then stored in the output relation *ApproximateJoin*. All the input/output relations are materialized in tables that are stored in a Relational Database Management System (RDBMS).

Listing 3.1: Example of a transformation wrote by the user.

```java
CREATE MATCHING ApproximateJoin
FROM Authors A, TeamAuthors T
LET sim = similarAuthors (A.firstname, A.lastname, T.firstname, T.lastname)
WHERE sim > 0.93 {
  SELECT
  T.teamid AS teamid,
  A.authorid AS authorid,
  T.firstname AS teamfirstname,
  T.lastname AS teamlastname,
  A.firstname AS authorfirstname,
  A.lastname AS authorlastname,
  sim
}
```

After the correct syntactical verification, for each data transformation, CLEENEX generates a Java class implementing each data transformation. This code generation is based on existing templates for each type of operator. Each class after its creation is then compiled.

2. **Execution:** The sequence of data transformations defined in the data cleaning program is executed in the same order as defined by the user. In other words, the Java classes, created and compiled in the compilation phase, are executed sequentially as defined in the Directed Acyclic Graph (DAG). The DAG is displayed to the user through a Graphical User Interface (GUI).

In Figure 3.1 we have the correspondent DAG of the example in Listing 3.1. The nodes of the DAG correspond to the data transformation (represented as an oval) and to the relations (represented
3.3 User interaction

When the user writes a data cleaning program in CLEENEX, he/she defines a set of data transformations to be applied to the dataset. However, since these transformations are typically applied to large datasets, it may not be possible to detect and correct all errors and inconsistencies automatically.

Since it is very difficult to consider all data quality problems when defining a data cleaning program, the user needs to revise several times the underlying logic of the data cleaning program. By doing this, the user is able to refine the data cleaning program in order to obtain more accurate results. However, it may exist very particular data quality problems that cannot be resolved automatically with the data cleaning program. The only way to overcome this limitation is for the user to manually correct any error or inconsistency that was not treated by the data cleaning program.

In order for the user to accomplish this kind of interaction in CLEENEX, the user must define Quality Constraints (QCs) and Manual Data Repairs (MDRs) [17]. QCs define the criteria that the records of each relation must obey. MDRs are a set of actions that a user can perform over a record that does not comply a given QC.

3.3.1 Quality Constraints

QCs are specified in a data cleaning program to ensure that the records comply with certain requirements after applying a given data transformation. Each relation in the data transformations graph can be associated to a set of QCs. The defined QCs can be applied either to the input relations, or to the output relation (i.e., after applying the transformation to the input relation). If a certain record does not satisfy a given QC, then that same record is marked as blamed and inserted into a table denominated as blamed tuples for the user to inspect it manually.

One example of a QC is $qc_1 : NOT \text{ NULL}(year)$, which means that the attribute $year$ cannot have a null value. By applying this QC to the relation in Figure 3.2, the first record will be marked as blamed, since its value in $year$ is null.
MDRs are used in a data cleaning program to provide the user a set of actions that he/she can perform in a given relation. A MDR is defined over a relation and consists in a \textit{View} and an \textit{Action}. A \textit{View} is a view over the relation, defined by the user, with the purpose to limit the amount of data to inspect. The \textit{Action} is a set of operations that can be applied to the \textit{View}. An \textit{Action} is either an insert, a delete or an update operation.

### 3.4 Component Architecture

Figure 3.3 shows the CLEENEX component architecture. It is constituted by the following components:

- **Parser**: It analyzes syntactically the specification of the data cleaning program defined by the user.
- **Catalog Manager**: Generates an internal representation, in Java, of the data cleaning program.
- **Database Manager**: Communicates with the Relational Database Management System (RDBMS). During the compilation step, it creates the output tables for each data transformation, as requested by the Catalog Manager. During the execution step, it executes SQL queries as requested by the Scheduler, the Debugger or the Graphical User Interface (GUI).
• **Quality Constraint (QC) Manager:** Parses and generates the corresponding Java code to enforce the user-defined QCs. This component is also responsible to create the blamed tuples table for each QC and sends the QCs information to the Manual Data Repair (MDR) Manager.

• **MDR Manager:** Parses the MDRs defined by the user. This component is also responsible to generate the corresponding Java code in order to construct and apply the units of user feedback based on those MDRs.

• **Scheduler:** Executes the compiled transformations according to their sequence in the data cleaning program.

• **GUI:** Constructs the graphical representation of the data cleaning graph.

• **Debugger:** Enables the user to trace an output record to its corresponding input records after applying a data transformation.

• **Optimizer:** For each logical data transformation, chooses the proper physical execution algorithms.

We will detail some of these components in order to better understand the core execution of CLEENEX. We will describe the **Parser**, which is responsible for analyzing and parsing the data cleaning programs written by the user. We will also explain how the **Optimizer** works in order to guarantee the optimal execution of the data cleaning program. The **Scheduler**, which is an important operational unit that executes the transformations that are specified in the data cleaning program. And finally, we will explain briefly the role of the **Debugger**, which allows the user to debug the data cleaning program.

### 3.4.1 Parser

The **Parser** is one of the operational units that has a role in the compilation phase. This component start by loading the file where the user wrote the data cleaning program. This file is then parsed and analyzed syntactically and semantically. After the correct parsing of the data cleaning program, CLEENEX generates the correspondent Java class for each transformation.

### 3.4.2 Optimizer

The **Optimizer** has a fundamental role regarding the data cleaning program efficiency. It is this component of CLEENEX that is responsible in deciding the best physical implementation for each transformation.

After the parsing of the data cleaning program, the **Optimizer** can choose the best physical implementation for each transformation. The **Optimizer** analyzes each data transformation and chooses if the execution of that transformation is either in Java or in SQL. After this, a new Java class is created. CLEENEX writes in this class the correspondent code for each transformation. After the code for each transformation is generated, all the created classes are then compiled in order to be executed.
3.4.3 Scheduler

This component is responsible to execute the data cleaning program written by the user. After the creation and compilation of all the Java classes for each transformation defined in the data cleaning program, the Directed Acyclic Graph (DAG) is created. When the user executes the data cleaning program through the GUI the compiled Java files are executed in the same order as in the DAG.

3.4.4 Debugger

When the user analyzes the results produced by the data cleaning program, sometimes he/she may find records whose errors/inconsistencies were not corrected. In other words, the user cannot determine how that record was produced in a given point of the DAG. Due to this situation, the user must know the origin of the errors/inconsistencies in a given record in order to make decisions to refine the data cleaning program.

CLEENEX allows the user to debug the data cleaning graph, by stepping through the graph forward and backward. This way, the user can understand the origin of the errors/inconsistencies in order to refine the transformations and thus correcting those errors/inconsistencies.
Chapter 4

Infrastructure for the CLEENEX Optimizer

As mentioned in Chapter 3, CLEENEX supports a clear separation between the logical and physical level. At the logical level, the user specifies the set of data transformations that compose the data cleaning program. The specification of the data cleaning program is through a language that extends SQL. Each data transformation that compose the data cleaning program is defined with a logical operator. CLEENEX supports five operators, in particular, the Match operator whose semantics corresponds to the approximate duplicate detection.

When the user specifies the match operator, he/she can invoke external functions that typically encode string matching algorithms. These algorithms are used within the Match operator to compare certain attributes of the input records in order to determine if a given pair of records is a match. The data cleaning logical specification is parsed and then at the physical level, it is chosen the most proper implementation for each operator.

As said in Chapter 3 for the match operator, CLEENEX always chooses to perform a Cartesian Product. However, if we are dealing with large datasets, then considering all possible combinations of tuples is not the best approach. In Chapter 2, we presented several alternatives to the Cartesian Product in order to improve the efficiency of the approximate duplicate detection.

Due to the clear separation between the logical and physical level, it is possible to choose different implementations for the match operator, without changing its logical definition. One way to be able to choose the most appropriate implementation of the match operator is with the use of an optimizer. The optimizer must be able to choose automatically, the best scale-up matching algorithm, having into account the logical specification of the match operator and the dataset that is being analyzed.

As we saw in Chapter 2, most of these algorithms that scale-up rule-based matching need some user-defined parameters (e.g., window size, thresholds, etc). The value of these parameters also have a great impact on the performance of a given algorithm both in terms of effectiveness and efficiency. The optimizer must also choose automatically which parameters are more suitable to be used along with the chosen scale-up matching algorithm.
Before we have a fully automated Optimizer with a good knowledge to make wise decisions, we first need to create an infrastructure in which the Optimizer will be implemented. This infrastructure will not have, however intelligence in order to know which is the best algorithm and the best parameter values for a given dataset. Since the Optimizer cannot make these decisions alone, it is fundamental for the user to provide hints to the optimizer that specify which scale-up algorithm and respective parameter values should be used to implement a given match operator.

In this Chapter, we detail the modifications that were applied to the CLEENEX architecture in order to build an infrastructure to implement an optimizer in CLEENEX. This infrastructure will support several alternatives to the Cartesian Product approach for the match operator and enable the user to choose how a given match operator should be implemented. This choice consists not only in selecting the algorithm to scale-up data matching, but also defining the parameters the user finds more adequate.

4.1 Solution Overview

In order to build an infrastructure to implement an optimizer for CLEENEX, we first need to analyze the current architecture of CLEENEX in order to determine which components we should modify/add. In the current version of CLEENEX, the optimizer has a minor impact in the efficiency of the data cleaning program. The user starts by writing the data cleaning program into a file using a specification language that extends SQL. This file is given to the Parser in order for the program to be syntactically analyzed and parsed.

One of the goals of this thesis is to allow the user to insert its decisions on how to optimize the matching transformation. To do so, we must add support for the user to provide hints to the optimizer in the specification of the data cleaning program. We must modify the Parser component of CLEENEX in order to allow support for the inclusion of hints in the data cleaning program. We also need to modify the internal representation of the transformations in order to store all the hints given by the user.

Currently, the optimizer always chooses the same predefined implementation for each type of transformation. In particular, the naive implementation of the matching transformation is by performing a Cartesian Product. After the definition of the physical implementations, CLEENEX generates and compiles a Java class for each transformation that corresponds to its implementation.

In order for the optimizer to have available alternatives to the Cartesian Product, we will implement several algorithms that scale-up rule-based matching. We will store these algorithms in the optimizer component. Since the Optimizer will not decide on its own which algorithm to choose, for each matching transformation, the decision will be made based on the hints given by the user during the specification of the data cleaning program. Having into account the decision made by the user, the matching transformation will be implemented with the correspondent physical algorithm. Besides the algorithm, the user can also suggest the parameter values that each algorithm requires. We need to modify the current Optimizer component in order to interpret the user’s hints. We also need to modify the code generation component, because the generated code will differ for each chosen scale-up matching algorithm.

Each transformation can be executed in Java or SQL. CLEENEX enables the execution of SQL
queries through the Java Database Connectivity (JDBC). The JDBC, is an Application Programming Interface (API) that allows connectivity between the Java programming language and a given database. In other words, it is possible to execute SQL queries directly to the Relational Database Management System (RDBMS) from the Java classes.

The match operator is currently implemented with a Cartesian Product and executed in Java. Since we can execute queries directly to the RDBMS through the JDBC, it is also possible to perform the Cartesian Product in SQL. However, since we use external functions to compute the similarity value for each pair of records we also need to call them inside the SQL query. The Oracle Database can store Java functions inside the RDBMS, so we need to add support for this RDBMS as well. This said, we will need to add support for the Oracle Database and modify any conflicting queries that are not supported by this RDBMS.

In each transformation it is possible for the user to invoke external functions. For the matching transformation, these functions can be string matching algorithms. However, in the current version of CLEENEX, there is only one string matching algorithm available. In order for the user to have more options and build more complex rules (i.e., that combines several string matching algorithms) we will need to add more string matching algorithms to the external functions library. This said, we need to modify the external functions library in order to enrich it with more string matching algorithms to be used by the matching transformations.

4.2 Library of External Functions

CLEENEX enables the invocation of external functions within the logical operators to transform data. For example, assuming we are defining a match operator, we can use a string matching algorithm to be used by transformation.

An external function is a method of a Java class. A class can hold several functions and typically the name of the class characterizes the type of functions it holds.

In this Section, we describe the string matching algorithms we added to the external functions library to be invoked by the match operator. In this Section we will also explain the reorganization that we performed to the library of external functions.

4.2.1 String matching algorithms

String matching algorithms are mainly used to compute the similarity value between each candidate pair in the matching transformation. The main goal of these algorithms is to determine, in a dataset, which record pairs correspond to the same entity. These string matching algorithms are stored in the external functions library. Currently, CLEENEX only has the Smith–Waterman algorithm [35], a sequence-based string matching algorithm.

In order to have more string matching algorithms available for the user, we developed and incorporated a Java implementation of the following algorithms (described with more detail in Chapter 2) in the
library of external functions:

- **Sequence-based**
  - Levenshtein distance
  - Jaro–Winkler distance

- **Token-based**
  - Jaccard similarity coefficient
  - Term Frequency/Inverse Document Frequency (TFIDF)

- **Phonetic**
  - Soundex
  - Metaphone

All of these algorithms accept as input two strings and return a float value that corresponds to the similarity value that goes between 0 and 1. The closer the computed value is to 1, the higher the similarity between those two strings.

**Implementation**

We implemented each of these algorithms and afterwards we performed sanity tests that allow us to guarantee that the implementation is correct. We made several tests that consisted in applying our string matching functions to various datasets, writing its results in a file and comparing with the results produced by other implementations of the same algorithms.

For the sequence-based and token-based string matching algorithms we used SecondString\(^1\) a Java-based package of approximate string matching techniques. We applied the equivalent string matching algorithms that we implemented and wrote the result into another file. For the phonetic string matching algorithms we used Apache Commons Codec\(^2\) a software that contains common encoders and decoders (e.g., Base64, Hex and Phonetic). The results produced by the Soundex and the Metaphone to the several datasets used by our string matching matching algorithms were also written into a file. We used the several datasets available in Febrl\(^3\) a software that does data standardization and probabilistic record linkage.

In the end we compared the files produced by our string matching algorithms with the files produced by SecondString and Apache Commons Codec. Since the output was the same, we can assume that our implementation is correct.

---

\(^1\)http://secondstring.sourceforge.net/
\(^2\)https://commons.apache.org
\(^3\)http://sourceforge.net/projects/febrl/
4.2.2 Reorganization of the external functions library

In the current version of CLEENEX, the classes of the external functions were in a single folder named \textit{functions}. This topology was somehow confusing since we have only one folder containing all the classes with the external functions that can be used by the various types of operators.

We re-arranged the \textit{functions} folder accordingly to the following packages:

- \textit{clustering}: Contains all the classes regarding clustering functions.
- \textit{datastructures}: Contains the classes that correspond to predefined types that can be used as parameters in functions or returned by functions.
- \textit{generateid}: Contains the several functions to generate IDs.
- \textit{merge}: Contains aggregation functions.
- \textit{standardize}: Has functions that normalize strings.
- \textit{stringmatching}: Contains string matching algorithms.
- \textit{extract}: Contains functions to parse the strings.
- \textit{exceptions}: Contains all the exceptions thrown by all the classes in the other packages.

Inside some of these packages we created a package named \textit{domainspecific} which contains the functions of that package category that are specific to a given dataset. We also added the package \textit{deprecated} that includes the classes that have functions that are not being used and older versions of a given class.

After defining the new organization, all the classes were analyzed in order to determine, from its functions, in which category they should be inserted. Some classes needed to be renamed, others merged and others split. Table 4.1 presents the modifications of those classes and the reason for the change.

We inserted all the classes into their corresponding folder in the new library of external functions. The classes that were not being used were assigned to the package \textit{deprecated}. In Figure 4.1 we have the final structure of the external functions library after the re-organization of the existing functions. We also added the new string matching algorithms that we implemented to the package \textit{stringmatching}.

We modified an existing data cleaning program according to the modifications that were made to the external functions. In Figure 4.2 we have the part of a data cleaning program for the \textit{CIDS} dataset. In Figure 4.2(a) we have the specification of the external functions signature before the modifications and in Figure 4.2(b) we have the same specification but after the modifications. As we can see, in Figure 4.2(b) the functions are more organized because their corresponding class has the name of the category of that function. This way it is easier to understand what a given function does.
4.3 Support for Oracle

Currently, CLEENEX executes the match operator with a Cartesian Product in Java (i.e., a two nested while). For each candidate pair of record, we apply a Java function that will compute the similarity value for that pair. If the produced value is equal to or greater than a value defined by the user, then the record pair matches. Each operator in CLEENEX can be executed in Java or in SQL. In order to allow the execution of the match operator in SQL, it is necessary to invoke the Java function inside the Relational Database Management System (RDBMS) in order to compute the similarity value of each candidate record pair.

One way to allow function calls inside the RDBMS is by utilizing Stored Procedures. Stored Procedures are Java methods stored in the database. In order to invoke these methods in a SQL query it is necessary to define the call specification for each Java method. This call specification maps the Java

Table 4.1: Modifications in the classes of the library of external functions.

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
<th>Change</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChoosesV3</td>
<td>ChooseCIDS</td>
<td>Name modification.</td>
<td>ChoosesV3 is the last version of the merge functions for the dataset CIDS. Since the older versions are in the package deprecated we can remove the version notation from the class name.</td>
</tr>
<tr>
<td>CidsAjax2</td>
<td>ExtractCIDS</td>
<td>Separation of the several functions from one existing class to newly created ones.</td>
<td>CidsAjax2 contains functions that belonged to different categories for the dataset CIDS.</td>
</tr>
<tr>
<td></td>
<td>StandardizeCIDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MergeCIDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DemoFunctions</td>
<td>MatchDemo</td>
<td>Name modification.</td>
<td>DemoFunctions contains matching functions for the dataset Demo.</td>
</tr>
<tr>
<td>Extract</td>
<td>ExtractCiteSeer</td>
<td></td>
<td>Extract contains extract functions for the dataset CiteSeer.</td>
</tr>
<tr>
<td>ExtractCoids</td>
<td>ExtractCIDS</td>
<td>Merged to existing class ExtractCIDS that has all the merge functions of the dataset CIDS.</td>
<td>ExtractCoids has more extract functions for the dataset CIDS.</td>
</tr>
<tr>
<td>Generate</td>
<td>GenerateID</td>
<td>Name modification.</td>
<td>Generate generates IDs.</td>
</tr>
<tr>
<td>MatchValue</td>
<td>MatchCiteSeer</td>
<td></td>
<td>MatchValue contains string matching functions that are used in standardization functions for the dataset CiteSeer.</td>
</tr>
<tr>
<td>Normal</td>
<td>StandardizeCiteSeer</td>
<td></td>
<td>Normal contains standardization functions for the dataset CiteSeer.</td>
</tr>
<tr>
<td>RegExp</td>
<td>RegExpCiteSeer</td>
<td></td>
<td>RegExp contains regular expressions that are used in functions of standardization for the dataset CiteSeer.</td>
</tr>
</tbody>
</table>

Figure 4.1: The new CLEENEX library of external functions.
The invocation of stored procedures in a SQL query is available for the Oracle Database, but CLEENEX currently runs only in the PostgreSQL Database. So, in order to execute the match operator in SQL, we need to add support for the Oracle RDBMS.

### 4.3.1 Implementation

As we said in Chapter 3, the core operation of CLEENEX consists in the compilation phase and in the execution phase. In the compilation phase the user writes the data cleaning program that is parsed. Afterwards, CLEENEX generates and compiles a Java class for each data transformation. In the execution phase, before executing each data transformation, CLEENEX drops all tables that were created in the previous execution. In the current version of CLEENEX we have the following definition for the drop
The if exists clause is not recognized by the Oracle RDBMS. One way to make the correspondent if exists clause in Oracle is in PL/SQL. PL/SQL is a programming language developed by Oracle as a procedural extension language for SQL and the Oracle relational database. This said, we needed to change the previous drop table query to the following PL/SQL procedure:

```plsql
begin
    execute immediate 'drop table TABLE_NAME';
exception
    when others then
        if sqlcode != -942 then
            raise;
        end if;
end;
```

The above PL/SQL procedure tries to drop the table with the name `TABLE_NAME`. If the table `TABLE_NAME` does not exist, then it will catch the "table not found" exception.

In the compilation phase, when CLEENEX compiles all the classes corresponding to the data transformations, it also compiles all the classes in the library of external functions. All the compiled functions are then aggregated in a Java Archive (JAR). Since we want to invoke these functions inside the RDBMS, we need to store them in the database. In order to store the Java functions in the Oracle RDBMS we must copy the JAR that contains all the compiled functions to the directory of the database. After this, we must load the Java functions by executing, in the RDBMS, the following command:

```
loadjava -user USER_NAME/USER_PASSWORD /DB_LOCATION/ajaxfunctions.jar
```

In the above command `USER_NAME` is the name of the user in the RDBMS, `USER_PASSWORD` is the password of the user, `DB_LOCATION` is the location of the database and `ajaxfunctions.jar` the JAR that contains all the compiled functions. In order to do this automatically, we added a script, that is executed after the compilation of the functions, that copies the JAR file to the location of the database and loads the functions to the RDBMS.

After loading the functions in the database we need to create the Java method names, the parameter types, and the return types used by each external function in the data cleaning program. CLEENEX supports simple types (e.g., `string`, `int`, etc) and composed types. A composed type is a structure that is composed by one or more types (simple or composed). For example, let us assume the following definition of a composed type in the data cleaning program:

```java
DEFINE COMPOSED TYPES AS CidsAuthor(name STRING(50), firstname STRING(20), lastname STRING(30));
```
In the above definition, we have the composed type CidsAuthor. CLEENEX after parsing this definition, converts into the following query:

```sql
create or replace type CidsAuthor force as object (name VARCHAR(50),
firstname VARCHAR(20), lastname VARCHAR(30))
```

The above query created an object with the same characteristics from the composed type defined in the data cleaning program. As we can see, the simple type (i.e., string) was directly converted to varchar.

After the creation of the composed types we need to create the functions signatures that correspond to the stored Java functions. Let us assume the following function signature defined in the data cleaning program:

```java
MatchCIDS.similarAuthors(STRING, STRING, STRING, STRING) RETURN FLOAT
```

This signature corresponds to the Java function similarAuthors in class MatchCIDS. The compiled Java class of this function is already stored in the RDBMS, so we just need to create the correspondent SQL function. CLEENEX creates the SQL functions, based in the Java functions signatures. Using the example of the function similarAuthors, CLEENEX converts into the following query:

```sql
create or replace function similarAuthorsStored(arg1 varchar, arg2 varchar, arg3 varchar, arg4 varchar)
return float
as language java
name MatchCIDS.similarAuthors(string, string, string, string) return float;
```

The above query creates the SQL function similarAuthorsStored that corresponds to the Java function similarAuthors. As we can see, all the arguments and return types of the Java function were converted to their correspondent SQL counterparts.

```sql
INSERT INTO ApproxJoin
SELECT * FROM ( SELECT T.teamid AS teamid, A.authorid AS authorid, T.firstname AS teamfirstname, T.lastname AS
teamlastname, A.firstname AS authorfirstname, A.lastname AS authorlastname,
similarAuthorsStored(A.firstname, A.lastname, T.firstname, T.lastname) AS sim FROM
GroupDistinctAuthors A, TeamAuthors T ) WHERE sim > 0.93
```

Listing 4.1: Example of a matching transformation.

In Listing 4.1, we have the query that is created when the matching transformation is implemented with a Cartesian Product in SQL. As we can see, the similarity function that is invoked in the insert query is similarAuthorsStored. When the similarAuthorsStored is called, the RDBMS executes its correspondent Java functions (i.e., similarAuthors).
4.4 Scale-up Matching Algorithms

Since one of the main focus is optimizing the matching transformation, the infrastructure needs to have scale-up matching algorithms as an alternative to the Cartesian Product. Since there are no available implementations of scale-up matching algorithms, we implemented in Java some scale-up matching algorithms (explained in detail in Chapter 2) and incorporated them in CLEENEX. The implementation of these algorithms are stored inside the package of the Optimizer.

4.4.1 Implementation

In order to determine which algorithms are worthier to implement, we analyzed a survey [6] from Peter Christen[4] of indexing techniques [5]. In this survey Peter Christen performed several tests to various algorithms that scale-up data matching, in order to measure their efficiency and effectiveness. After analyzing the results of these experiments, we verified that the best trade-off between efficiency and effectiveness were obtained by the following six algorithms:

- **Traditional Blocking**
- **Traditional Sorted-Neighborhood Join (SNJ)**
- **Multi-pass approach**
- **Adaptive SNJ approach**
- **SNJ Inverted Index**
- **Canopy Clustering**

We implemented all these algorithms in Java, because CLEENEX is also implemented in Java. After the implementation of these algorithms we needed to make sure that they could encounter all the possible candidate pairs. To do so, we performed sanity tests with two datasets. In Appendix A we have available all the sanity tests that have been done and their corresponding results.

After performing the sanity tests to the scale-up algorithms, we incorporated in CLEENEX in order to have a set of several algorithms available for the Optimizer. This way, there are several alternatives to implement the Match operator instead of performing the Cartesian Product. We gathered all the scale-up algorithms we implemented and added to a package inside of the Optimizer package.

4.5 Hints

This infrastructure to implement the Optimizer cannot make decisions on its own on how to optimize the matching transformation. However, the user can provide a hint to the optimizer in the data cleaning program specification in order to inform the Optimizer how a given matching transformation should be implemented.

4.5.1 Specification Language

We have defined several types of hints and corresponding values that can be used in the specification of the data cleaning program. One example of a type of hint is $scale-up$, where its value will correspond to the scale-up algorithm to be used to implement a given transformation. For example, if the user writes $scale-up = "Traditional Blocking"$ when specifying a matching transformation, then that transformation will be implemented with the $Traditional Blocking$ algorithm.

All the algorithms that scale-up data matching that were implemented, need a user-defined key. This key can be composed by one attribute value (i.e., simple) or can be formed by parts of one or more attributes values (i.e., composed). One type of hint we defined was the $scale-up key$ that allows the user to specify which key should be used by the algorithm. If the user wants to use a simple key, he/she just needs to specify the name of the record field. If the user wants to specify a composed key, he/she must use the following syntax:

$$(\text{first | last}) N \text{ from ATTRIBUTE\_NAME} , (\text{first | last}) N \text{ from ATTRIBUTE\_NAME})*$$

In the above definition, the fist token corresponds if we want the key to be formed with the first or with the last characters of the attribute $\text{ATTRIBUTE\_NAME}$. The $N$ value specifies the number of characters that we want from the attribute $\text{ATTRIBUTE\_NAME}$. If we want to add more characters from other attributes in the same key, we can just put a comma and specify the characters we want to append. For example, if we define $scale-up key = "first 2 from a, last 3 from b"$, then it means the key is composed by the 2 first characters from the value of the attribute $a$ and with the last 3 characters from the value of the attribute $b$.

Every algorithm that scale-up data matching that was implemented supports the composed keys. During the execution of the algorithm to scale-up data matching, if the composed key syntax is detected, then the algorithm creates the composed key. This composed key will be created according to the specification made by the user in the hint $scale-up key$.

We defined several hint types that can be combined with other hint types. In Table 4.2, we have all the types of hints CLEENEX supports on the first column and their corresponding values in the second column. The third column is the restrictions of each hint. For example, the hint $implementation$ can have two types of values: Java or SQL. If the user chooses SQL he/she cannot use the hint $scale-up$, because only the $Cartesian Product$ can be executed in SQL. If the $implementation$ value is Java, then the user can optionally choose any scale-up algorithm with the hint $scale-up$. If the user specifies $implementation = Java$ and does not specify any value for the hint $scale-up$, then the matching transformation will be performed with a $Cartesian Product$ executed in Java.

According to the chosen scale-up algorithm, the user can then specify the correspondent parameters of that algorithm. Of course the user must have a previous knowledge about the algorithm he/she choses in order to provide the most adequate parameters. For example, if the user chooses $scale-up = Multi-pass$, he/she must specify at least two keys with the hint $scale-up keys$ rather than only one with the hint $scale-up key$. 

43
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL</td>
<td>Only Cartesian Product</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scale-up</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Blocking</td>
<td>Traditional SNJ</td>
<td>implementation=Java</td>
</tr>
<tr>
<td>Traditional SNJ</td>
<td>Multi-pass</td>
<td></td>
</tr>
<tr>
<td>Inverted Index SNJ</td>
<td>Adaptive SNJ</td>
<td></td>
</tr>
<tr>
<td>Canopy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scale-up Key</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any attribute or composed key</td>
<td>scale-up=Traditional Blocking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Traditional SNJ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Inverted Index SNJ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Adaptive SNJ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Canopy</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scale-up Keys</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any attributes and composed keys</td>
<td>scale-up=Multi-pass</td>
<td>At least two attributes delimited by &amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Measure</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard</td>
<td>scale-up=Adaptive SNJ</td>
<td></td>
</tr>
<tr>
<td>TF-IDF</td>
<td>scale-up=Adaptive SNJ</td>
<td></td>
</tr>
<tr>
<td>Levenshtein</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jaro-Winkler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Soundex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metaphone</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any double</td>
<td>scale-up=Adaptive SNJ</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Window</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any integer</td>
<td>scale-up=Traditional SNJ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Multi-pass</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scale-up=Inverted Index SNJ</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tight Threshold</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any double</td>
<td>scale-up=Canopy</td>
<td>Higher than loose threshold</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loose Threshold</th>
<th>Value</th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any double</td>
<td>scale-up=Canopy</td>
<td>Lower than tight threshold</td>
</tr>
</tbody>
</table>

Table 4.2: Type of hints, correspondent values and restrictions.

### 4.5.2 Implementation

In order to add support to hints, we modified the *Parser* component to parse the hint specification in the data cleaning program.

In Listing 4.2 we have after the FROM clause the hint token represented as a % (this token represents a comment). In the hint definition we can specify the type of hints. In this example we have three hint types: `scale-up`, `scale-up key` and `window`. This hint specifies that the transformation `ApproxJoin` will be implemented with the *Traditional Sorted-Neighborhood Join (SNJ)* algorithm, using the attribute `lastname` as a key and using 5 as the size of the sliding window.

```sql
CREATE MATCHING ApproxJoin
FROM GroupDistinctAuthors A, TeamAuthors T
% scale-up="Traditional SNJ" scale-up key = "lastname" window = 5%
LET sim = similarAuthors (A.firstname, A.lastname, T.firstname, T.lastname)
WHERE sim > 0.93 {
SELECT
T.teamid AS teamid,
A.authorid AS authorid,
T.firstname AS teamfirstname,
T.lastname AS teamlastname,
A.firstname AS authorfirstname,
A.lastname AS authorlastname,
sim }
```

Listing 4.2: Example of a hint in a matching transformation.

During the parsing of the data cleaning program, if the parser encounters any hint defined in the
matching transformation then CLEENEX stores internally in a structure that associates the type of hint and its value. As we can see in Figure 4.5, the class CleaningParser, during the parsing of the data cleaning programs, also parses the hints and stores them in the transformation in the init() method.

4.6 Code generation

After the correct parsing of the data cleaning program, for each data transformation, a Java file is created. This file will contain the implementation of each transformation to be later executed in the execution phase. The implementation of each transformation is given by the Optimizer component. Since we added the support to hints, we need to modify the Optimizer component in order to interpret the provided hints to make the decision.

As we can see in Figure 4.5, in the class Optimizer, when determining if the implementation is Java or SQL, the Optimizer reads the hint in the transformation. Based on the value of that hint the Optimizer defines the implementation accordingly. In the class Matching, after the initialize() method, the Java file is created. The code that is written into that file depends on the value in the hint type scale-up.

Implementation

For each scale-up algorithm and for the Cartesian Product (in Java or SQL), the produced code will differ. In Listing 4.3, we have an example of the code generated of a matching transformation implemented with the Cartesian Product. As we can see, all the records from tables GroupDistinctAuthors and TeamAuthors are stored in a ResultSet. A ResultSet contains all the resulting records from a given query. In this example, theResultA and theResultB will contain all the records from tables GroupDistinctAuthors and TeamAuthors, respectively. The transformation computes the candidate records pairs by performing a Cartesian Product. For each candidate record pair, it is computed the similarity value directly from each record in the ResultSet.

```java
wm.openTable("GroupDistinctAuthors", wm.JDBC_TABLE);
ResultSet theResultA = wm.executeQuery("select * from GroupDistinctAuthors");
wm.openTable("TeamAuthors", wm.JDBC_TABLE);
ResultSet theResultT = wm.executeQuery("select * from TeamAuthors");
while (theResultA.next()) {
    theResultT.beforeFirst();
    while (theResultT.next()) {
        try {
            float sim = 0;
            sim = MatchCIDS.similarAuthors(theResultA.getString(2), theResultA.getString(3), theResultT.getString(3), theResultT.getString(4));
        }
    }
}
```

Listing 4.3: Part of the generated code for a matching transformation.

When using scale-up algorithms we need to have a structure that can hold both the record values and its key. So, we made a structure that is a Java class named Item that is stored in the package structures inside the package of the Optimizer. The Item is used by the scale-up algorithms and contains the
key value and an object Record. A Record is another structure stored in the package structures. Each instance of this class will contain the data of a given record and the table name from where the record came.

```java
wm.openTable("GroupDistinctAuthors", wm.JDBC_TABLE);
ResultSet theResultA = wm.executeQuery("select * from GroupDistinctAuthors");
wm.openTable("TeamAuthors", wm.JDBC_TABLE);
ResultSet theResultT = wm.executeQuery("select * from TeamAuthors");
for (Map<String, Record> pair : TraditionalBlocking.execute("A", theResultA, "T", theResultT, "firstname")) {
    try {
        float sim = 0;
        sim = MatchCIDS.similarAuthors(pair.get("A").getContent("firstname"), pair.get("A").getContent("lastname"), pair.get("T").getContent("firstname"), pair.get("T").getContent("lastname"));
    }
}
```

Listing 4.4: Part of the generated code for a matching transformation.

In Listing 4.4 we have the excerpt of the same transformation but using the Traditional Blocking algorithm. As we can see both ResultSet are inserted into the algorithm along with the attribute to be used as a key. Inside of the scale-up algorithm it will be created an instance of Item for each record in the ResultSet. After the structure creation the algorithm executes and returns a list with all the candidate Record pairs. For each pair it is computed the similarity measure with the external function.

In both cases (Cartesian Product or scale-up algorithm), if the produced similarity measure complies the where clause defined by the user, then that pair is inserted into the output table.

4.7 Integration with the GUI

It was being developed, in parallel with this thesis, a graphical editor [13] for CLEENEX, with the purpose to replace the previous CLEENEX Graphical User Interface (GUI).

The new GUI allows the user to select the external functions by category in the code editor window used to edit the specification code. When the user is writing a specific transformation and starts to write the LET clause, the GUI shows a pop-up with the various categories available in the library of external functions. The user selects the category according to the transformation. In particular, in the matching transformation, the user can choose the stringmatching category, in order to use the available string matching algorithms.

In Figure 4.3 we have a print-screen of the GUI when defining the LET clause. As we can see, a pop-up appears that shows to the user the various categories inside the external function library. If the user, for example, chooses the category stringmatching it will be displayed the name of the classes that are inside that category.

Regarding the hints, the user can also define them during the creation of the data cleaning program. The new GUI also shows a pop-up when the user writes the HINT clause (i.e., %). The GUI shows the user the various types of hints he can choose (e.g., implementation, scale-up key, etc) and the type
of values he/she can choose. For example, if the user chooses 'scale-up', the GUI shows a list of the available algorithms the user can select to implement the match operator. In the case of thresholds, window sizes and keys, the user writes the value he/she finds more appropriate. In Figure 4.4 we have a matching transformation with the hints. As we can see, the syntax of the hints is the same as it was in the old version.
Figure 4.5: Sequence diagram of the generation of the file for a matching transformation.
Chapter 5

Validation

In this Chapter we describe the experiments that we conducted to validate the implemented infrastructure. The infrastructure allows the user to choose the most appropriate algorithm for each dataset. This experimental validation allows us to see the results produced by several algorithms for each dataset in terms of efficiency and effectiveness.

An analysis that we have done is related to the algorithms’ parameters. Each scale-up algorithm has specific parameters that the user must insert (e.g., window size, similarity measures, etc). For each dataset, we evaluated the behavior of each scale-up algorithm with different parameters, in terms of effectiveness and efficiency. In the end we analyzed the results in order to understand which parameters for each scale-up algorithm are better suited for each dataset.

5.1 Experimental Setup

This section describes the datasets we used, the matching criteria we used to find approximate duplicates and the metric we evaluated.

The experiments were conducted on a computer with 8 cores with 2.4GHz and 32 Gigabytes of main memory (RAM). The system runs GNU/Linux 3.13.0-48 (Ubuntu 14.04.3) and uses Java 1.8.0._60.

5.1.1 Datasets

We used the following datasets:

- **CDDB**: The dataset **CDDB** includes 9763 CD records randomly extracted from [freeDB](http://www.freedb.org/) This dataset contains 298 duplicates.

- **DatasetA**: The **DatasetA** contains informations about persons. This dataset was generated with [dsgen](http://sourceforge.net/projects/febrl/), a dataset generator available in [Febrl](http://sourceforge.net/projects/febrl/) a software that supports data standardization and probabilistic record linkage. This dataset contains 10000 records (5000 original and 5000 duplicates), with exactly one duplicate per original record and one modification (in one field) per record.
CDDB

The CDDB dataset is described by the following attributes:

- **did**: This attribute corresponds to the CD unique identification number.
- **artist**: This attribute contains the name of the artist of the CD.
- **title**: This attribute contains the title of the CD.
- **category**: This attribute contains the category of the disk.
- **track\(_X\)**: This attribute contains the track name of the CD, where \(X\) corresponds to the track number.

DatasetA

The DatasetA is described by the following attributes:

- **rec.id**: This attribute contains the record’s unique identification number.
- **given.name**: This attribute contains the first name of the person.
- **surname**: This attribute contains the last name of the person.
- **street.number**: This attribute contains the street number where the person lives.
- **address.1**: This attribute contains the first address of the person.
- **address.2**: This attribute contains the second address of the person.
- **suburb**: This attribute contains the name of the suburb where the person lives.
- **postcode**: This attribute contains the postal code where the person lives.
- **state**: This attribute contains the state from where the person is.
- **date.of.birth**: This attribute contains the date of birth of the person.
- **age**: This attribute contains the age of the person.
- **phone.number**: This attribute contains the phone number of the person.
- **soc.sec.id**: This attribute contains the Social Security Number (SSN) of the entity.

5.1.2 Metrics

In this section, we describe the metrics we used to evaluate the scale-up matching algorithms. In particular, we evaluated their efficiency and effectiveness.
Effectiveness

The effectiveness is highly related to the matching quality. In other words, since we are dealing with rule-based matching, the effectiveness depends on the user-defined rules. Besides of the rules, the algorithm’s parameters (e.g., window size, key, thresholds, etc) can have an impact on the effectiveness of the algorithm. For example, if the user chooses a bad key for that dataset it may not generate the best candidate pairs and will thus find few approximate duplicate record pairs.

To evaluate the effectiveness of the scale-up matching algorithms we used the following three measures: (i) precision; (ii) recall; (iii) f-measure. In these three measures we consider the number of correctly identified duplicates, the number of identified duplicates and the number of true duplicates.

\[
\text{precision} = \frac{\text{correctlyIdentifiedDuplicates}}{\text{identifiedDuplicates}} \quad (5.1)
\]

In equation 5.1 we have defined the formula to measure precision. Precision is a common measure to apply when dealing with algorithms of duplicate detection, since it measures the proportion of true matches detected among all records that were identified as match (correctly or not). Basically, it measures how precise a given algorithm is on identifying true matches.

\[
\text{recall} = \frac{\text{correctlyIdentifiedDuplicates}}{\text{trueDuplicates}} \quad (5.2)
\]

In equation 5.2 we have defined the formula of recall. Similar to accuracy, recall measures the portion of records that were correctly identified as true matches among all existing true matches in the dataset. In other words, it measures how many true matches have been correctly identified.

\[
\text{f-measure} = 2 \times \left( \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \right) \quad (5.3)
\]

In equation 5.3 we have defined the formula of f-measure. The f-measure combines both precision and recall. In order to have an acceptable f-measure, it must exist a good balance between recall and precision (both must have high values).

Efficiency

One metric that we used to evaluate the efficiency of each scale-up matching algorithm is the number of comparisons. Most of the computational cost is on the comparison between the records, so in order for the algorithm to be efficient it must minimize the number of comparisons. This measure depends on the amount of records in the dataset and in the parameters that were provided (such as the window size, key value, etc).

Additionally, we measured the execution time of each algorithm. In other words, we measured how long an algorithm takes to detect the approximate duplicates in a given dataset.
5.2 Scale-up Matching Algorithms

For each dataset we tested the following scale-up algorithms:

1. **Traditional Blocking**: In this algorithm we varied between the two keys.

2. **Traditional Sorted-Neighborhood Join (SNJ)**: Besides the keys, we used two different window sizes: 5 and 10. We performed tests for each key.

3. **Multi-pass Approach**: In this algorithm we used the same window sizes used in the tests for the Traditional SNJ. We used both keys.

4. **Adaptive SNJ Approach**: In this algorithm we used several similarity measures (i.e., string matching algorithms). We also used two different thresholds: 0.65 and 0.85. These tests were made for each key.

5. **Inverted Index SNJ**: We varied between the same window sizes used in the tests for the Traditional SNJ. During these tests we also tried for the two different keys.

6. **Canopy Clustering**: The similarity measure we used in the tests are two token-based string matching algorithms: Jaccard and Term Frequency/Inverse Document Frequency (TFIDF). The two pairs of thresholds (loose and tight) we used in the tests were loose threshold $= 0.65$; tight threshold $= 0.85$ and loose threshold $= 0.8$; tight threshold $= 0.9$. We conducted the tests for each user-defined key.

All these scale-up matching algorithms need a user defined key as a parameter. In order to compare the impact of the chosen key on each result, we defined two keys for each dataset. We defined one simple key (i.e., a key formed only by one attribute) and a composed key (i.e., a key formed with one or more attributes). We defined the following keys for each dataset:

- **CDDDB**
  - **Composed key**: This key is formed with the first 4 characters from the attribute `artist` and with the last 3 characters from the attribute `title`.
  - **Simple key**: This key is the value in the attribute `title`.

- **DatasetA**
  - **Composed key**: This key is formed with the first 3 numbers from the attribute `phone_number` and with the last 2 numbers from the attribute `soc_sec_id`.
  - **Simple key**: This key is the value in the attribute `surname`.

5.3 Matching Rules

We prepared several rules (specific for each dataset) to evaluate their accuracy and coverage. We performed the Cartesian Product in order to obtain the pairs of records that satisfy each rule.
5.3.1 CDDB

We analyzed the dataset and verified that the most relevant fields in order to build a rule was using the artist and the title attributes. For this dataset we defined the following rules:

- **Rule 1:** Jaro-Winkler(artist) ≥ 0.9 ∧ Jaccard(title) ≥ 0.9
- **Rule 2:** Metaphone(artist) ≥ 0.9 ∧ Term Frequency/Inverse Document Frequency (TFIDF)(title) ≥ 0.9
- **Rule 3:** Jaccard(artist) ≥ 0.9 ∧ Levenshtein(title) ≥ 0.9

We applied the Cartesian Product to the CDDB dataset for each rule. After each execution of the Cartesian Product we collected the number of matches (and respective pairs) and verified how many of those pairs were false matches. This verification could be done because the dataset provides documentation that indicates which records are true matches. We verified the list of pairs of records produced by the Cartesian Product and marked the one that were not in the documentation of the dataset as false matches.

<table>
<thead>
<tr>
<th></th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>227</td>
<td>225</td>
<td>221</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>18</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>Precision</td>
<td>0.921</td>
<td>0.92</td>
<td>0.928</td>
</tr>
<tr>
<td>Recall</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.959</td>
<td>0.958</td>
<td>0.963</td>
</tr>
</tbody>
</table>

Table 5.1: Number of matches and false matches of the CDDB dataset.

In Table 5.1 we have the number of matches and false matches for each rule. We compared the produced matches for each rule in order to choose the rule we would use during the experiments to the scale-up algorithms. We used the first rule, since it encountered more approximate duplicate pairs than the other rules. We used the matches and the false matches produced by this rule as reference during the experiments to the scale-up algorithms. Since the number of matches was 227 and the number of false matches was 18, the number of true duplicates of this dataset is 209 (i.e., 227 − 18).

5.3.2 DatasetA

We analyzed the dataset and based on examples of records that belonged to the same person we defined the rules. For this dataset we defined the following rules:

- **Rule 1:** Jaro-Winkler(given_name) ≥ 0.9 ∧ Jaro-Winkler(surname) ≥ 0.9 ∧ Levenshtein(soc_sec_id) ≥ 0.9
- **Rule 2:** Jaccard(suburb) ≥ 0.9 ∧ TFIDF(phone_number) ≥ 0.9
- **Rule 3:** Metaphone(state) ≥ 0.9 ∧ TFIDF(street_number) ≥ 0.9 ∧ JaroWinkler(postcode) ≥ 0.9
Table 5.2: Number of matches and false matches of the datasetA dataset.

Since these datasets were generated with dsgen, there is no documentation in order for us to verify which are the correct matches. However, we know that each record has exactly one duplicate.

In Table 5.2 we have the results for each rule. As we can see, none of the rules encountered all true duplicates. However, the first rule covered more records pairs than the other rules. This said, we used the first rule to perform all the tests to the algorithms to scale-up data matching.

5.4 Effectiveness Experimental Results

In this section we present the results of the performed tests regarding effectiveness. For each dataset we tested each scale-up matching algorithm. For each experiment performed to the algorithms, we collected the number of matches and the number of false matches. With these values we computed the precision, the recall, and the f-measure.

5.4.1 CDDB

• Traditional Blocking: In Table 5.3 we have the results obtained with the Traditional Blocking using each key. As we can see, the difference of the results is not very significant. The Traditional Blocking using the simple key is slightly better than using the composed key just because it encounters one more pair. This algorithm only finds around 93% of record pairs comparing to the Cartesian Product.

<table>
<thead>
<tr>
<th></th>
<th>Composed key</th>
<th>Simple key</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>211</td>
<td>212</td>
<td>227</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>10</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>Precision</td>
<td>0,953</td>
<td>0,948</td>
<td>0,921</td>
</tr>
<tr>
<td>Recall</td>
<td>0,962</td>
<td>0,962</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0,957</td>
<td>0,955</td>
<td>0,959</td>
</tr>
</tbody>
</table>

Table 5.3: Results after applying the Traditional Blocking to the CDDB dataset.

• Traditional Sorted-Neighborhood Join (SNJ): In Table 5.4 we have the results for the tests performed for the Traditional SNJ. For this algorithm, besides trying for the two keys, we also used the two window sizes. Once again the difference was not very significant. Comparing with the Traditional Blocking, this algorithm encountered more pairs, specially when using the simple key. The size of the window did not affect at all the results. This algorithm finds around 99% of the record pairs comparing to the Cartesian Product.

54
• **Multi-pass:** In the Multi-pass algorithm we used both keys and tried for two different window sizes. In Table 5.5, we have the results after executing the Multi-pass algorithm. As we can see, this algorithm encountered the same record pairs as the Cartesian Product. However, the window size did not affect the results, since it produced the same pairs. Which means that even with a small window size (i.e., 5), this algorithm encountered all the records because it combined the independent results of applying the Traditional SNJ for each two keys. Although performing the Traditional SNJ with a simple key and a window size = 5 resulted in 225 record pairs, it still did not encountered two record pairs. However, this record pairs were found when using the composed key with the same window size. That's why when we combine both results we encounter all the record pairs as the Cartesian Product.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>218</td>
<td>218</td>
<td>225</td>
<td>226</td>
<td>227</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Precision</td>
<td>0.936</td>
<td>0.936</td>
<td>0.92</td>
<td>0.92</td>
<td>0.921</td>
</tr>
<tr>
<td>Recall</td>
<td>0.976</td>
<td>0.976</td>
<td>0.99</td>
<td>0.995</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.955</td>
<td>0.955</td>
<td>0.954</td>
<td>0.956</td>
<td>0.959</td>
</tr>
</tbody>
</table>

Table 5.4: Results after applying the Traditional SNJ to the CDDB dataset.

• **Inverted Index SNJ:** In the Inverted Index SNJ algorithm, we also tried, for each key, two different window sizes. In Table 5.6, we have the results after executing the Inverted Index SNJ. As we can see, the results are not different when using 5 or 10 as a window size.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>219</td>
<td>219</td>
<td>226</td>
<td>226</td>
<td>227</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Precision</td>
<td>0.936</td>
<td>0.936</td>
<td>0.92</td>
<td>0.92</td>
<td>0.921</td>
</tr>
<tr>
<td>Recall</td>
<td>0.981</td>
<td>0.981</td>
<td>0.995</td>
<td>0.995</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.958</td>
<td>0.958</td>
<td>0.956</td>
<td>0.956</td>
<td>0.959</td>
</tr>
</tbody>
</table>

Table 5.5: Results after applying the Multi-pass to the CDDB dataset.

Table 5.6: Results after applying the Inverted Index SNJ to the CDDB dataset.

In comparison to the Traditional SNJ, it encounters more pairs because the Inverted Index SNJ uses an inverted index table. In the inverted index table, the index keys are all the unique key values generated from all records. For each inverted index key we will have an inverted index list with all the records that generated that key value. In each window slide, regardless of the window size, the algorithm will compare all records with the same key value, because each unique key only appears once in the inverted index table. This approach encounters around 99% comparing to the Cartesian Product and the Multi-pass approach.
• **Adaptive SNJ:** In the Adaptive SNJ, after the sorting phase, the algorithm uses string matching algorithms to apply to the key value of each adjacent record. The algorithm compares the records’ key value (after being sorted by the key value) and if the value is equal or greater than a defined threshold then the window size increases. In the end there will be several windows (with different window sizes) that will hold the similar records. In the tests for this algorithm we experimented with the following string matching algorithms: (i) Jaccard; (ii) Term Frequency/Inverse Document Frequency (TFIDF); (iii) Levenshtein; (iv) Jaro-Winkler and (v) Metaphone. We ran experiments for each of the two key values and for each string matching algorithm we tried two different thresholds.

We conducted experiments for the Adaptive SNJ algorithm with the various configurations (i.e., string matching algorithms, thresholds and key values). In order to minimize, we are only showing, for each string matching algorithm, the configurations that had better results in the tests.

In Table 5.7 we have the best results after conducting the experiments to the Adaptive SNJ algorithm. In all the configurations, the simple key had better results than using the composed key. The best results for this algorithm is when we use the Jaccard algorithm with a threshold of 0.65 or the Jaro-Winkler algorithms with a threshold of 0.85. The quality of the Adaptive SNJ algorithm relies heavily on the similarity measure (and threshold) that is chosen for the user-defined key.

<table>
<thead>
<tr>
<th>Simple key Measure = Jaccard Thresholds = 0.65</th>
<th>Simple key Measure = TFIDF Thresholds = 0.65</th>
<th>Simple key Measure = Levenshtein Thresholds = 0.85</th>
<th>Simple key Measure = Jaro-Winkler Thresholds = 0.85</th>
<th>Simple key Measure = Metaphone Thresholds = 0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>226</td>
<td>213</td>
<td>225</td>
<td>226</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0.92</td>
<td>0.94</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>Precision</td>
<td>221</td>
<td>210</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>Recall</td>
<td>0.956</td>
<td>0.955</td>
<td>0.956</td>
<td>0.954</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.956</td>
<td>0.955</td>
<td>0.956</td>
<td>0.954</td>
</tr>
</tbody>
</table>

Table 5.7: Best results after applying the Adaptive SNJ to the CDDB dataset.

• **Canopy:** In the Canopy algorithm we conducted tests using the following string matching algorithms: (i) Jaccard and (ii) TFIDF. For both string matching algorithms we used two different pairs of thresholds.

In Table 5.8 we have the results after all the tests performed with the Canopy algorithm. The best results were in the configurations that used the simple key. As we can see in Table 5.8 the interval of thresholds that produced better results was 0.65 and 0.85. The interval of the thresholds influences the effectiveness of the algorithm, because, the wider the interval, the more records will fall into several canopies as other records. If the interval is very tight, then the probability of the record to be in only one or two canopies is higher.

<table>
<thead>
<tr>
<th>Simple key Measure = Jaccard Thresholds = 0.65 and 0.85</th>
<th>Simple key Measure = TFIDF Thresholds = 0.65 and 0.85</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>221</td>
<td>227</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>Precision</td>
<td>0.932</td>
<td>0.921</td>
</tr>
<tr>
<td>Recall</td>
<td>0.986</td>
<td>0.957</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.958</td>
<td>0.954</td>
</tr>
</tbody>
</table>

Table 5.8: Best results after applying the Canopy algorithm to the CDDB dataset.
5.4.2 DatasetA

- **Traditional Blocking:** In Table 5.9 we have the results after executing the *Traditional Blocking* for both keys. As we can see, the composed key found much more pairs of records than the simple key. However, the algorithm only found around 89% of the record pairs comparing to the *Cartesian Product*.

<table>
<thead>
<tr>
<th></th>
<th>Composed key</th>
<th>Simple key</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 447</td>
<td>4 026</td>
<td>4 563</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.975</td>
<td>0.882</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.987</td>
<td>0.937</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.9: Results after applying the *Traditional Blocking* algorithm to *datasetA*.

- **Traditional SNJ:** In Table 5.10 we have the results for the tests performed for the *Traditional SNJ*. We tried, for each key, two different window sizes. For the composed key, the results with 5 as window size, were slightly worse than the *Traditional Blocking*. This happened because the window size was too small. With the same key, but with a bigger window (i.e., 10), the algorithm encountered around 98% of record pairs comparing to the *Cartesian Product*. For the simple key the results were always worse than the *Traditional Blocking*. Even with 10 as a window size, the algorithm only finds around 70% of the record pairs, comparing to the *Cartesian Product*. One solution to find more record pairs with this algorithm using the simple key, is by using a bigger window size. However, this will result in a decrease on the performance of the algorithm since more candidate record pairs will be generated.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 283</td>
<td>4 449</td>
<td>2 621</td>
<td>3 206</td>
<td>4 563</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.939</td>
<td>0.975</td>
<td>0.574</td>
<td>0.703</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.969</td>
<td>0.987</td>
<td>0.729</td>
<td>0.826</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.10: Results after applying the *Traditional SNJ* algorithm to *datasetA*.

- **Multi-pass:** In Table 5.11 we have the results for the *Multi-pass* approach. We used the same window sizes as the *Traditional SNJ*. As we can see, with 10 as window size, the algorithm found around 99% of record pairs comparing to the *Cartesian Product*.

<table>
<thead>
<tr>
<th></th>
<th>Window = 5</th>
<th>Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 432</td>
<td>4 538</td>
<td>4 563</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.971</td>
<td>0.995</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.985</td>
<td>0.997</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.11: Results after applying the *Multi-pass* algorithm to *datasetA*.
In Table 5.12 we have the results for the tests performed for the Inverted Index SNJ. We used the same window size as the ones during the tests with the Traditional SNJ. As we can see, with 10 as window size, the algorithm encountered more record pairs. This algorithm, comparing to the Cartesian Product, encountered around 98% record pairs.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 454</td>
<td>4 474</td>
<td>4 502</td>
<td>4 542</td>
<td>4 563</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.976</td>
<td>0.98</td>
<td>0.987</td>
<td>0.995</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.968</td>
<td>0.99</td>
<td>0.983</td>
<td>0.997</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.12: Results after applying the Inverted Index SNJ algorithm to datasetA.

In Table 5.13 we have the best results for the tests performed for the Adaptive SNJ with the various configurations. In order to minimize, we are only showing the best results for the performed tests. Comparing to the Cartesian Product, this algorithm can find around 98% of the record pairs. In this dataset, the best results belonged to the composed key, except for the Metaphone algorithm because we did not even made tests with this algorithm and the composed key. Since the composed key is formed only with numbers, the use of the Metaphone algorithm to compare the key values does not make sense.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Measure = Jaccard Threshold = 0.65</th>
<th>Composed key Measure = TFIDF Threshold = 0.85</th>
<th>Composed key Measure = Levenshtein Threshold = 0.85</th>
<th>Composed key Measure = Jaro-Winkler Threshold = 0.85</th>
<th>Simple key Measure = Metaphone Threshold = 0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 456</td>
<td>4 447</td>
<td>4 447</td>
<td>4 458</td>
<td>4 187</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.977</td>
<td>0.975</td>
<td>0.975</td>
<td>0.977</td>
<td>0.916</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.986</td>
<td>0.987</td>
<td>0.987</td>
<td>0.988</td>
<td>0.956</td>
</tr>
</tbody>
</table>

Table 5.13: Best results after applying the Adaptive SNJ algorithm to datasetA.

In Table 5.14 we have the best results of the tests performed for the Canopy algorithm. This algorithm, comparing to the Cartesian Product can find around 98% record pairs.

<table>
<thead>
<tr>
<th></th>
<th>Composed key Measure = Jaccard Thresholds = 0.8 and 0.9</th>
<th>Composed key Measure = TFIDF Thresholds = 0.65 and 0.85</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of matches</td>
<td>4 450</td>
<td>4 447</td>
<td>4 563</td>
</tr>
<tr>
<td>Number of false matches</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.975</td>
<td>0.975</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.987</td>
<td>0.987</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.14: Best results after applying the Canopy algorithm to datasetA.

5.5 Efficiency Experimental Results

In this section we will present the results of the performed tests regarding efficiency. For each dataset we tested each algorithm to scale-up data matching. For each algorithm we also tried different configurations (e.g., window size, thresholds, etc) in order to understand how it impacts the performance of
each algorithm.

In order to evaluate the efficiency of each algorithm we performed each test five times in order to calculate the minimum, the maximum and the average run-time of each algorithm. Besides this, we also collected the number of comparisons that were made in each algorithm. In order to analyze the efficiency gain compared to the Cartesian Product we also measured its efficiency for each dataset.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>CDDB</th>
<th>DatasetA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>Minimum</td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>47 653 203</td>
<td>590 153</td>
</tr>
<tr>
<td></td>
<td>49 995 000</td>
<td>743 786</td>
</tr>
</tbody>
</table>

Table 5.15: Results of the efficiency tests performed by the Cartesian Product for each dataset.

In Table 5.15 we have the run-times of the Cartesian Product for each dataset. As we can see, the run-time is related to the number of candidate record pairs that are generated. As said in Chapter 1, each comparison has an associated cost, so the more records there are to compare, the more time it will take.

5.5.1 CDDB

- Traditional Blocking: In Table 5.16 we have the results after executing the Traditional Blocking for both keys. As we can see, comparing with the Cartesian Product, the number of comparisons and the time is highly significantly since we gain around 97% of efficiency.

<table>
<thead>
<tr>
<th>Composed key</th>
<th>Simple key</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comparisons</td>
<td>16 926</td>
<td>15 458</td>
</tr>
<tr>
<td>Minimum</td>
<td>484</td>
<td>306</td>
</tr>
<tr>
<td>Maximum</td>
<td>645</td>
<td>395</td>
</tr>
<tr>
<td>Average</td>
<td>543,2</td>
<td>336</td>
</tr>
</tbody>
</table>

Table 5.16: Results after applying the Traditional Blocking to the CDDB dataset.

- Traditional Sorted-Neighborhood Join (SNJ): In Table 5.17 we have the results after executing the Traditional SNJ for both keys. We used two different window sizes: 5 and 10. Although the results are still better than the Cartesian Product we can verify that the bigger the window size is, the bigger the complexity of the algorithm. For both keys, the number of comparisons is always bigger for the window size = 10, resulting in bigger execution times. Comparing to the Cartesian Product we have a huge gain in terms of the average run-time.

<table>
<thead>
<tr>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comparisons</td>
<td>39 038</td>
<td>39 038</td>
<td>39 038</td>
<td>47 653 203</td>
</tr>
<tr>
<td>Minimum</td>
<td>87 813</td>
<td>87 813</td>
<td>87 813</td>
<td>590 153</td>
</tr>
<tr>
<td>Maximum</td>
<td>7 189</td>
<td>32 391</td>
<td>7 084</td>
<td>31 638</td>
</tr>
<tr>
<td>Average</td>
<td>32 548,4</td>
<td>7 151,8</td>
<td>31 914</td>
<td>604 329,6</td>
</tr>
</tbody>
</table>

Table 5.17: Results after applying the Traditional SNJ to the CDDB dataset.
• **Multi-pass**: In Table 5.18 we have the results after executing the *Multi-pass* with the both user-defined keys. We used two different window sizes: 5 and 10. Like in the *Traditional SNJ*, the efficiency of the algorithm is highly dependent on the size of the window. For bigger windows, more candidate record pairs are generated, and thus are made more comparisons. However, comparing to the *Cartesian Product*, the algorithm had, in average, a better run-time.

<table>
<thead>
<tr>
<th></th>
<th>Window = 5</th>
<th>Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comparisons</td>
<td>87 128</td>
<td>196 844</td>
<td>47 653 203</td>
</tr>
<tr>
<td>Time (ms)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>32 443</td>
<td>152 505</td>
<td>590 153</td>
</tr>
<tr>
<td>Maximum</td>
<td>32 727</td>
<td>153 371</td>
<td>624 316</td>
</tr>
<tr>
<td>Average</td>
<td>32 562,2</td>
<td>152 992,8</td>
<td>604 329,6</td>
</tr>
</tbody>
</table>

Table 5.18: Results after applying the *Multi-pass* to the CDDB dataset.

• **Inverted Index SNJ**: In Table 5.19 we have the results after executing the *Inverted Index SNJ* for each key. We also used the window sizes 5 and 10 during these tests. Like in the previous SNJ variants, the window size affects the efficiency of the algorithm. Comparing to the *Cartesian Product*, we have better run-time values.

<table>
<thead>
<tr>
<th></th>
<th>Composed key</th>
<th>Simple key</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>72 747</td>
<td>130 105</td>
<td>56 851</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>71 478</td>
<td>181 593</td>
<td>48 243</td>
</tr>
<tr>
<td>Maximum</td>
<td>74 653</td>
<td>198 029</td>
<td>49 163</td>
</tr>
<tr>
<td>Average</td>
<td>73 246</td>
<td>191 767</td>
<td>48 831,2</td>
</tr>
</tbody>
</table>

Table 5.19: Results after applying the *Inverted Index SNJ* to the CDDB dataset.

• **Adaptive SNJ**: In Table 5.20 we have the best results after executing the *Adaptive SNJ*. In these algorithms the produced number of candidate record pairs depends on the similarity measure and on the threshold value that are chosen by the user. The configuration that produced less candidate record pairs was using the *Levenshtein* algorithm as similarity measure with 0.85 as threshold. The similarity measure that produced more candidate records pairs was the *Metaphone* algorithm. Comparing to the *Cartesian Product*, these algorithms had better average run-time.

<table>
<thead>
<tr>
<th></th>
<th>Simple key</th>
<th>Simple key</th>
<th>Simple key</th>
<th>Simple key</th>
<th>Simple key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comparisons</td>
<td>Measure = Jaccard</td>
<td>Measure = Term Frequency/Inverse Document Frequency (TFIDF)</td>
<td>Measure = Levenshtein</td>
<td>Measure = Jaro-Winkler</td>
<td>Measure = Metaphone</td>
</tr>
<tr>
<td>Minimum</td>
<td>1 367</td>
<td>13 577</td>
<td>17 150</td>
<td>19 150</td>
<td>26 256</td>
</tr>
<tr>
<td>Maximum</td>
<td>14 108</td>
<td>19 150</td>
<td>27 077</td>
<td>27 077</td>
<td>32 999</td>
</tr>
<tr>
<td>Average</td>
<td>3 450</td>
<td>1 577,2</td>
<td>3 170,8</td>
<td>3 170,8</td>
<td>3 323</td>
</tr>
</tbody>
</table>

Table 5.20: Best results after applying the *Adaptive SNJ* to the CDDB dataset.

• **Canopy**: In Table 5.21 we have the best results after executing the *Canopy* algorithm. As we can see, in the configuration with the *TFIDF* algorithm we have more candidate record pairs than the ones resulting in the configuration with the *Jaccard* algorithm. Comparing to the *Cartesian Product*, this algorithms shows better results in the average time.

60
### 5.5.2 DatasetA

- **Traditional Blocking**: In Table 5.22 we have the results after the several executions of the *Traditional Blocking* for both keys. As we can see we have a significant gain in terms of efficiency comparing to the *Cartesian Product*. Most of this gain is due to the reduction of the number of comparisons that are made throughout the execution of the algorithm. Comparing with the *Cartesian Product*, the algorithm has a gain of around 93%.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>Composed key</th>
<th>Simple key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>15 150</td>
<td>198 174</td>
</tr>
<tr>
<td>Maximum</td>
<td>164</td>
<td>464</td>
</tr>
<tr>
<td>Average</td>
<td>192,4</td>
<td>520,4</td>
</tr>
</tbody>
</table>

Table 5.22: Results after applying the *Traditional Blocking* to DatasetA.

- **Traditional SNJ**: In Table 5.23 we have the results after the tests to the *Traditional SNJ*. Once more, there is a significant gain in terms of efficiency comparing to the *Cartesian Product*. The number of comparisons increases according to the size of the window. The run-time of the algorithm increases proportionately with the number of candidate record pairs. Comparing with the *Cartesian Product* we have a significant gain in performance.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
</tr>
<tr>
<td>Maximum</td>
</tr>
<tr>
<td>Average</td>
</tr>
</tbody>
</table>

Table 5.23: Results after applying the *Traditional SNJ* to DatasetA.

- **Multi-pass**: In Table 5.24 we have the results after the tests to the *Multi-pass* approach using both keys. As we can see, when using 5 as window, the number of comparisons is lower, comparing to the same configuration but with 10 as window size. The *Multi-pass* approach, after combining the results from the independent runs of the *Traditional SNJ*, performs the *Transitive Closure*. The *Transitive Closure* applies a *Cartesian Product* to the resulting record pairs in order to find more record pairs.

- **Inverted Index SNJ**: In Table 5.25 we have the results after applying the *Inverted Index SNJ* using different window sizes. When using the simple key, the algorithm has a large loss on performance. This happens because the value of the chosen key is not very discriminate and a lot of records
are inserted in the same index on the inverted index table. Comparing with the *Cartesian Product*, depending on the configuration, the *Inverted Index SNJ* can be more efficient.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>Window = 5</th>
<th>Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>77 354</td>
<td>176 534</td>
<td>49 995 000</td>
</tr>
<tr>
<td>Maximum</td>
<td>523 585</td>
<td>537 969</td>
<td>761 142</td>
</tr>
<tr>
<td>Average</td>
<td>453 665,8</td>
<td>524 155,4</td>
<td>754 358,5</td>
</tr>
</tbody>
</table>

Table 5.24: Results after applying the *Multi-pass* to *DatasetA*.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>Composed key Window = 5</th>
<th>Composed key Window = 10</th>
<th>Simple key Window = 5</th>
<th>Simple key Window = 10</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>140 212</td>
<td>295 798</td>
<td>343 456</td>
<td>535 008</td>
<td>49 995 000</td>
</tr>
<tr>
<td>Maximum</td>
<td>191 518</td>
<td>513 531</td>
<td>1 515 167</td>
<td>4 454 096</td>
<td>761 142</td>
</tr>
<tr>
<td>Average</td>
<td>188 091,8</td>
<td>776 547,5</td>
<td>1 541 538,8</td>
<td>4 132 728,4</td>
<td>754 358,5</td>
</tr>
</tbody>
</table>

Table 5.25: Results after applying the *Inverted Index SNJ* to *DatasetA*.

- **Adaptive SNJ**: In Table 5.26 we have the best results after the tests performed to the *Adaptive SNJ*. The results were better when using the *Jaro-Winkler* algorithm with 0.85 as threshold because it produced less candidate record pairs. However, even with few candidate record pairs, this algorithm encountered around 98% of the record pairs, comparing to the *Cartesian Product*.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>Composed key Measure = Jaccard Threshold = 0.65</th>
<th>Composed key Measure = TFIDF Threshold = 0.85</th>
<th>Composed key Measure = Levenshtein Threshold = 0.85</th>
<th>Composed key Measure = Jaro-Winkler Threshold = 0.85</th>
<th>Simple key Measure = Metaphone Threshold = 0.85</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>134 719</td>
<td>150 967</td>
<td>129 970</td>
<td>73 931</td>
<td>125 343</td>
</tr>
<tr>
<td>Maximum</td>
<td>147 022</td>
<td>165 719</td>
<td>130 144</td>
<td>72 947</td>
<td>130 144</td>
</tr>
<tr>
<td>Average</td>
<td>137 254,8</td>
<td>161 957,6</td>
<td>129 729</td>
<td>73 859,2</td>
<td>129 729</td>
</tr>
</tbody>
</table>

Table 5.26: Best results after applying the *Adaptive SNJ* with the simple key to *DatasetA*.

- **Canopy**: In Table 5.27 we have the best results after the tests to the *Canopy* algorithm. With this composed key, the results were significantly better when using the *TFIDF* algorithm because it resulted in less comparisons and hence in a better run-time.

<table>
<thead>
<tr>
<th>Number of comparisons</th>
<th>Composed key Measure = Jaccard Thresholds = 0.8 and 0.9</th>
<th>Composed key Measure = TFIDF Threshold = 0.65 and 0.85</th>
<th>Cartesian Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>40 077</td>
<td>15 150</td>
<td>49 995 000</td>
</tr>
<tr>
<td>Maximum</td>
<td>188 958</td>
<td>175 708</td>
<td>743 786</td>
</tr>
<tr>
<td>Average</td>
<td>198 823</td>
<td>184 600</td>
<td>761 142</td>
</tr>
</tbody>
</table>

Table 5.27: Best results after applying the *Canopy* algorithm to *DatasetA*.

### 5.6 Discussion

In Figure 5.1 we have the overall effectiveness of the algorithms to scale-up *data matching*. For each dataset and for each algorithm we chose the algorithm with the best relation between the *f-measure* and the average run-time. These chosen algorithms are highlighted in bold in the previous tables that
presented the results for the various configurations for each algorithm. We built this graph in order to compare all the scale-up matching algorithms for each dataset with the *Cartesian Product*. As we can see, in general, the effectiveness of the algorithms to scale-up data matching does not vary significantly, comparing to the *Cartesian Product*.

Figure 5.1: Overall effectiveness of the scale-up matching algorithms and the *Cartesian Product*.

For the *CDDB* dataset, the best scale-up matching algorithm, in terms of effectiveness, belongs to the *Multi-pass* algorithm with both keys and 5 as window size. This algorithm had an *f-measure* equal to the *Cartesian Product*, since it found the same record pairs. All the other approaches had lower *f-measure* values. For the *DatasetA* dataset, the best algorithm, in terms of effectiveness, belongs to the *Adaptive Sorted-Neighborhood Join (SNJ)* with the composed key, using the Jaro-Winkler as similarity measure and 0.85 as threshold. This algorithm, however, did not found the same record pairs as the *Cartesian Product*, but it found around 98% of the record pairs.

In Figure 5.2 we have the overall efficiency of the algorithms to scale-up data matching. We used the average time of the same algorithms (and corresponding configurations) from the previous graph. We also collected the average time of the *Cartesian Product* and built a graph in order to compare the average times (in milliseconds) of all approaches. As we can see, there is a noticeable difference in the average time values according to the algorithms to scale-up data matching.

Figure 5.2: Overall efficiency of the scale-up matching algorithms and the *Cartesian Product*.

For the *CDDB* dataset, the best algorithm, in terms of efficiency, belongs to the *Traditional Blocking* algorithm with the composed key, since its average time was lower than the other approaches. This happened due to the simplicity of the *Traditional Blocking* algorithm, since it only blocks the records
and compares the records inside the same block. In terms of effectiveness, this algorithm had a lower f-measure than the Cartesian Product. The reason for the lower f-measure is due to the limitation of the Traditional Blocking algorithm. If the record fields that form the key have errors in its values, then the produced key will be different from the key of other records that belong to same real entity. Due to this, these records will never fall into the same block and thus compared. However, the difference of the effectiveness is not very significant and comparing to the gain in performance over the Cartesian Product it may be better to use the Traditional Blocking.

For the CDDB dataset, the Canopy was the algorithm that had worse results in terms of efficiency. However, in terms of effectiveness, this algorithm had a pretty good f-measure value, comparing to the other approaches. There is a trade-off in using more complex algorithms to accomplish more accurate results because the efficiency decreases. In terms of effectiveness, the Multi-pass approach was the best algorithm for this dataset. In terms of efficiency this algorithm, comparing to the Cartesian Product, reduced around 82% of comparisons.

In conclusion, for the CDDB dataset, the Multi-pass algorithm with both keys and 5 as window size, was better in terms of the relation between effectiveness and efficiency.

For the DatasetA dataset, the best algorithm, in terms of efficiency, also belongs to the Traditional Blocking with the composed key. Once again, this algorithm did not have the best results in terms of effectiveness. However, the difference is not very significant and compared to the gain in performance (around 98%) over the Cartesian Product, it may be more profitable to use the Traditional Blocking.

For the DatasetA dataset, the Multi-pass approach with both keys and 5 as window size was the worst algorithm, in terms of efficiency. This happened due to the dataset size. Since it performs twice the Traditional SNJ and combines both results. Afterwards the algorithm applies the Transitive Closure, thus taking more time.

In general, more complex approaches (i.e., Multi-pass and Canopy) produce more accurate results, however they take more time to execute. For big datasets it may be more wise to choose more simpler approaches, since the gain in performance will overcome the slight loss in accuracy. Another conclusion that we can take from this analysis is that the Multi-pass approach is a good algorithm to use only if the independent runs of the Traditional SNJ produce a small amount of record pairs. The reason why we conclude this is because the Transitive Closure performs a Cartesian Product in the resulting record pairs found in the independent runs of the Traditional SNJ for each key. For example, in the dataset CDDB, the Multi-pass algorithm before applying the Transitive Closure encountered around 227 record pairs. Applying a Cartesian Product to this number of records is relatively quick. In the DatasetA dataset, the Multi-pass approach, before applying the Transitive Closure encountered around 4432 record pairs. Performing a Cartesian Product to this set is more computationally expensive and, given the effectiveness of this algorithm to this dataset, is not advantageous to use it.
Chapter 6

Conclusions

In this document we developed an infrastructure to support the implementation of the CLEENEX optimizer. This infrastructure provides several algorithms to scale-up data matching in order to make the matching transformation more efficient. In order for the user to choose which algorithm should implement a given matching transformation we added hints to the specification of the data cleaning program. With hints, the user can specify, during the creation of the transformation, which algorithm (and correspondent parameters) should be used to improve the efficiency of the data cleaning process.

In this document we also explained several approaches that are used to address the challenge of effectiveness and efficiency in data matching. In particular, we explained record matching techniques and algorithms to scale-up data matching. Record matching techniques are used to find approximate duplicate records accurately. Algorithms to scale-up data matching are approaches used in order to find approximate duplicate records efficiently.

In this Chapter we summarize the work done during the creation of the infrastructure for the optimizer component in CLEENEX. In this Chapter we also describe the future work of this thesis.

6.1 Summary

In Chapter 1 we introduced to data cleaning problems, in particular the existence of approximate duplicate records. In this Chapter we also explained the notion of data cleaning, that aims at correcting data quality problems. The detection of approximate duplicates is the process that corrects the existence of approximate duplicate records. This process, however, has some challenges regarding effectiveness and efficiency. In this Chapter we referred some data cleaning tools that focus on correcting data quality problems. However, each tool has a fixed implementation for each type of data transformation. This may result in a decrease of performance of the data cleaning process if the implementation is not the most suited for the dataset that is being analyzed. In this Chapter we presented the objectives of this thesis, that consists in creating an infrastructure to implement an optimizer. Finally, we described the structure of this document.

In Chapter 2 we described some approaches to address the challenges regarding effectiveness and
efficiency of the data matching process. In particular, for effectiveness, we described two approaches that are used to address this challenge: string matching algorithms and rule-based matching. In order to improve efficiency of the approximate duplicate detection, we described algorithms to scale-up data matching.

In Chapter 3, we have explained CLEENEX, the data cleaning tool that we used to implement the infrastructure. The reason why we chose this data cleaning tool is because of its clear separation of the logical specification and the physical implementation of the data transformations. This separation allows to change the implementation without affecting the logical specification of the data transformation, thus guaranteeing an efficient execution of the data cleaning process. In this Chapter we described the current architecture and core operation of CLEENEX.

In Chapter 4, we explained all the modifications that were made to CLEENEX in order to build the infrastructure for the optimizer component. We also explained the algorithms we implemented (i.e., string matching algorithms and algorithms to scale-up data matching) and respective sanity tests. In this Section we also showed the final result after the integration with the new Graphical User Interface (GUI).

In Chapter 5, we described the validation made for each algorithm to scale-up data matching. We presented the methodology we used regarding the efficiency and effectiveness of each algorithm. In the end we analyzed each result in order to compare, for each dataset, which algorithm had a better performance when detecting the approximate duplicate records.

In conclusion, we developed an infrastructure that allows the implementation of the CLEENEX optimizer. With this infrastructure, the user can easily choose which algorithm implements a given match operator in order to result in the most efficient execution of the approximate duplicate detection.

6.2 Future Work

In order to continue the work of this thesis, we propose the following tasks for future work:

- The current infrastructure does not have the knowledge to make decisions on how to optimize the matching transformation. This means that the user must be the one who makes the decision on how a given matching transformation should be optimized. However, an optimizer should be able to do this automatically. One task is to implement this mechanism that allows the optimizer to choose automatically which algorithm to scale-up data matching should be used. Besides choosing the algorithm the optimizer should also be able to choose the most suited parameters for the chosen algorithm having into account the type of dataset. One idea is to collect a sample of the dataset that is being analyzed. With this sample, the Optimizer tries different predefined approaches in order to measure the efficiency and effectiveness of each approach. In the end, the Optimizer chooses the algorithm that had a better trade-off regarding both measures. This algorithm is then used to optimize the matching transformation.

- The current hint type scale-up key supports the definition of simple keys (i.e., keys that are formed with one attribute value) and composed keys (i.e., keys that are formed with parts of one or more
attributes values). However, the syntax to define composed keys only supports the use of the first or last \( n \) characters of a given attribute value. The syntax of the composed keys could support more types of character extraction to form composed keys. For example, consider the following definition of a scale-up key: \[ \text{scale-up key} = \text{extractNumbers}(a) \], where \( \text{extractNumbers} \) is a function that extracts the numbers present in the value of the attribute \( a \). This way we can combine various types of character extraction to form the composed keys. We propose the implementation of more functions to extract characters in a string in order to form composed keys.
Bibliography


Appendix A

Sanity Tests for Scale-up Matching Algorithms

This Appendix describes the various sanity tests that we performed for each scale-up matching algorithm and the correspondent results. It starts with a brief description of the dataset and its attributes, followed by the rules applied to each candidate record pair and the various configurations that were used in each test.

For each dataset we tested the following scale-up algorithms:

1. Traditional Blocking
2. Traditional Sorted-Neighborhood Join (SNJ)
3. Multi-pass Approach
4. Adaptive SNJ Approach
5. Inverted Index SNJ
6. Canopy Clustering

A.1 Overall Setup

A.1.1 Window size

The window size is a parameter that Traditional Sorted-Neighborhood Join (SNJ), Multi-pass Approach and Inverted Index SNJ need. For all the datasets we used two window sizes:

1. 5
2. 20

As we can see, we tried both for a window size relatively small, and for a bigger window size.
A.1.2 Similarity measures

Both Adaptive SNJ Approach and Canopy Clustering use similarity measures in order to determine window sizes and compute clusters, respectively. We used several string matching algorithms, namely:

1. Jaccard
2. Term Frequency/Inverse Document Frequency (TFIDF)
3. Levenshtein
4. Jaro-Winkler
5. Metaphone

For the Adaptive SNJ Approach we used all the string matching algorithms (except Metaphone when the key was composed by numbers). For the Canopy Clustering we also used Jaccard and TFIDF, because the algorithm only uses cheap Token-based string matching algorithms.

A.1.3 Thresholds

Both Adaptive SNJ Approach and Canopy Clustering need to have thresholds defined. The Adaptive SNJ Approach needs one threshold, and the Canopy Clustering needs two thresholds (a tight threshold and a loose threshold).

We tested with two thresholds for the Adaptive SNJ Approach:

1. 0.85
2. 0.65

For the Canopy Clustering we tested with two pairs of thresholds:

1. • Tight threshold: 0.85
   • Loose threshold: 0.65
2. • Tight threshold: 0.9
   • Loose threshold: 0.8

As we can see the second pair of thresholds has a smaller margin between the two values.

A.2 Cora Dataset

The dataset Cora consists in 1295 distinct citations to 122 documents from the Cora research paper search engine. It contains the following attributes:

- **author**: This attribute contains the author of the publication.
A.2.1 Setup

Rules

We analyzed the dataset and defined a rule that covered some record pairs that belonged to the same real entity. For this dataset we defined the following rule:

1. In order to consider a pair of records as match, the records must have:
   
   - At least a similarity value of 0.9 when applying the Jaccard string matching algorithm to the title field;
   - At least a similarity value of 0.9 when applying the Term Frequency/Inverse Document Frequency (TFIDF) string matching algorithm to the venue field;
   - At least a similarity value of 0.9 when applying the Levenshtein string matching algorithm to the author field;
   - Have the same value in the year field;

Keys

For this dataset we defined two keys, which are:

1. author field;
2. title field;
A.2.2 Results

We performed the Cartesian Product to the Cora dataset and the algorithm returned a total of 1177 approximate duplicates.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Key 1</th>
<th>Key 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1019</td>
<td>1156</td>
</tr>
</tbody>
</table>

Table A.1: Results of the Traditional Blocking tests.

On Table A.1 we have the results after applying the Traditional Blocking to the same dataset with the same rule. Since Traditional Blocking only compares records that share the same key, some of the records that are matches, but do not have the same key, are never compared.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Key 1 &amp; Window 1</th>
<th>Key 1 &amp; Window 1</th>
<th>Key 1 &amp; Window 1</th>
<th>Key 1 &amp; Window 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>661</td>
<td>1146</td>
<td>411</td>
<td>1034</td>
</tr>
</tbody>
</table>

Table A.2: Results of the Traditional SNJ tests.

On Table A.2 we have the results after applying the Traditional Sorted-Neighborhood Join (SNJ) to the Cora dataset. As we can see, if the window is too small there will be less pairs of records found. However, the bigger the window size, the bigger the complexity of the algorithms, since there will be more comparisons in order to determine if a given pair is a match. However, even with a window size of 20 not all pairs were found.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Keys &amp; Window 1</th>
<th>Keys &amp; Window 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>995</td>
<td>1146</td>
</tr>
</tbody>
</table>

Table A.3: Results of the Multi-pass tests.

On Table A.3, we have the results after applying the Multi-pass algorithm. For this algorithm we used both keys author and title. As we can see, even with a small window size we can discover much more record pairs than with the Traditional SNJ. This is due to the fact that there is an union of the independent runs of the Traditional SNJ. Besides, after this union, the algorithm applies a Transitive Closure in order to discover even more pairs.

<table>
<thead>
<tr>
<th>Window 1</th>
<th>Key 1</th>
<th>Key 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1152</td>
<td>1171</td>
</tr>
<tr>
<td>Window 2</td>
<td>1156</td>
<td>1172</td>
</tr>
</tbody>
</table>

Table A.4: Results of the SNJ Inverted Index tests.

On Table A.4 we have the results of the SNJ Inverted Index tests. As we can see, it discovers more pairs than the Traditional SNJ. However it does not encounter all pairs, because of the window size.

On Table A.5 and Table A.6 we have the results after applying the Adaptive SNJ with key 1 and 2, respectively. As we can see the results vary with the similarity measure chosen by the user.

On Table A.7 we have the results after applying the Canopy algorithm. As we can see, with this algorithm we can obtain the same number of pairs as performing the Cartesian Products.
Table A.5: Results of the Adaptive SN tests with attribute author as key.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Threshold 1</th>
<th>Threshold 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure 1</td>
<td>1149</td>
<td>1151</td>
</tr>
<tr>
<td>Measure 2</td>
<td>1019</td>
<td>1019</td>
</tr>
<tr>
<td>Measure 3</td>
<td>1151</td>
<td>1152</td>
</tr>
<tr>
<td>Measure 4</td>
<td>1153</td>
<td>1153</td>
</tr>
<tr>
<td>Measure 5</td>
<td>1152</td>
<td>1152</td>
</tr>
</tbody>
</table>

Table A.6: Results of the Adaptive SNJ tests with attribute title as key.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Threshold 1</th>
<th>Threshold 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure 1</td>
<td>1168</td>
<td>1171</td>
</tr>
<tr>
<td>Measure 2</td>
<td>1156</td>
<td>1156</td>
</tr>
<tr>
<td>Measure 3</td>
<td>1168</td>
<td>1171</td>
</tr>
<tr>
<td>Measure 4</td>
<td>1171</td>
<td>1172</td>
</tr>
<tr>
<td>Measure 5</td>
<td>1172</td>
<td>1172</td>
</tr>
</tbody>
</table>

Table A.7: Results of the Canopy tests.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Key 1 &amp; Thresholds 1</th>
<th>Key 1 &amp; Thresholds 2</th>
<th>Key 2 &amp; Thresholds 1</th>
<th>Key 2 &amp; Thresholds 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure 1</td>
<td>1177</td>
<td>1174</td>
<td>1177</td>
<td>1177</td>
</tr>
<tr>
<td>Measure 2</td>
<td>1019</td>
<td>1019</td>
<td>1157</td>
<td>1156</td>
</tr>
</tbody>
</table>

A.3 Restaurant Dataset

The dataset Restaurant has 864 records with the name and address of restaurants from Fodor’s and Zagat’s guidebooks. In this dataset exists 112 duplicates. It contains the following attributes:

- **name**: This attribute contains the name of the restaurant.
- **addr**: This attribute contains the address of the restaurant.
- **city**: This attribute contains the city of the restaurant.
- **phone**: This attribute contains the phone of the restaurant.
- **type**: This attribute contains the type of restaurant.
- **class**: This attribute contains an identification code that is shared by the same restaurant. This parameter is only used to confirm if two records correspond to the same restaurant.

A.3.1 Setup

Rules

We analyzed the datasets and defined rules that covered some record pairs that belonged to the same real entity. For this dataset we defined two rules, which were:

1. In order to consider a pair of records as match, the records must have:
   - At least a similarity value of 0.9 when applying the Term Frequency/Inverse Document Frequency (TFIDF) string matching algorithm to the name field;
• At least a similarity value of 0.9 when applying the Levenshtein string matching algorithm to the *phone* field;

2. In order to consider a pair of records as match, the records must have:

• At least a similarity value of 0.9 when applying the Metaphone string matching algorithm to the *name* field;

• At least a similarity value of 0.9 when applying the Jaccard string matching algorithm to the *addr* field;

• At least a similarity value of 0.85 when applying the Jaro-Winkler string matching algorithm to the *city* field;

**Keys**

For this dataset we defined two keys, which are:

1. *phone* field;

2. *name* field;

### A.3.2 Results

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>65</td>
</tr>
</tbody>
</table>

Table A.8: Results of the *Cartesian Product* execution.

On Table A.8 we have the results after applying the *Cartesian Product* to the *Restaurant* dataset for both rules.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 1</td>
<td>80</td>
</tr>
<tr>
<td>Key 2</td>
<td>77</td>
</tr>
</tbody>
</table>

Table A.9: Results of the *Traditional Blocking* tests.

On Table A.9 we have the results after applying the *Traditional Blocking* to the *Restaurant* dataset. As we can see, the pairs of records that were covered by the first rule share the same value for the attribute *phone*. However, for the second rule, there are some record pairs that have the value of the *phone* attribute different. For the attribute *name* as key, in none of the rules the values are the same.

<table>
<thead>
<tr>
<th>Rule 1 &amp; Key 1</th>
<th>Rule 1 &amp; Key 2</th>
<th>Rule 2 &amp; Key 1</th>
<th>Rule 2 &amp; Key 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window 1</td>
<td>80</td>
<td>77</td>
<td>65</td>
</tr>
<tr>
<td>Window 2</td>
<td>80</td>
<td>77</td>
<td>65</td>
</tr>
</tbody>
</table>

Table A.10: Results of the *Traditional SNJ* tests.

On Table A.10 we have the results for the *Traditional Sorted-Neighborhood Join (SNJ)*. As we can see, the window size for this dataset does not affect significantly the results. The main reason why in
In some cases the algorithm did not encounter the same pairs is because of the key values. In the sorting phase of the Traditional SNJ, the records are ordered by their key value. Since some of these records have errors in their key values, when ordered, they will not be close to each other. When the sliding window starts to compare the records, similar records will not fall under the same window because they are far away. One solution would be to increase the window size, but it would also increase the complexity of the algorithm, since it would compare more records inside each iteration of the sliding window. At some point, the complexity of the algorithm would not compensate the results.

Table A.11: Results of the Multi-pass tests.

<table>
<thead>
<tr>
<th>Rule 1 &amp; Keys</th>
<th>Rule 2 &amp; Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window 1</td>
<td>80</td>
</tr>
<tr>
<td>Window 2</td>
<td>80</td>
</tr>
</tbody>
</table>

On Table A.11 we have the results after applying the Multi-pass algorithm. As we can see all the pairs were discovered since this algorithm made an union for the independent results for each key. The Traditional SNJ with the attribute phone as key already encounters all the pairs as performing the Cartesian Product. This said, the union with the results of the Traditional SNJ with the attribute name as key turns out to be somewhat pointless, because it will not find any new record pairs. The same happens with the seconds rule, except that for a small window size (i.e., 5), the Multi-pass approach can find the record pair that was not found when applying the Traditional SNJ with the attribute name as key.

Table A.12: Results of the SNJ Inverted Index tests.

<table>
<thead>
<tr>
<th>Rule 1 &amp; Key 1</th>
<th>Rule 1 &amp; Key 2</th>
<th>Rule 2 &amp; Key 1</th>
<th>Rule 2 &amp; Key 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window 1</td>
<td>80</td>
<td>77</td>
<td>65</td>
</tr>
<tr>
<td>Window 2</td>
<td>80</td>
<td>77</td>
<td>65</td>
</tr>
</tbody>
</table>

On Table A.12 we have the results after applying the SNJ Inverted Index to the Restaurant dataset. In this case, unlike for the Cora dataset, this algorithm did not find more pairs than the Traditional SNJ. In fact, this algorithm produced exactly the same number of pairs. This happens because of the same reason as in the Traditional SNJ. Although this algorithm uses an inverted index that in fact compares more records in each iteration than the Traditional SNJ, since the similar records are far-away from each other they will never fall under the same window to be compared.

Table A.13: Results of the Adaptive SNJ tests for the first rule with attribute phone as key.

<table>
<thead>
<tr>
<th>Measure 1</th>
<th>Measure 2</th>
<th>Measure 3</th>
<th>Measure 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold 1</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Threshold 2</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>

Table A.14: Results of the Adaptive SNJ tests for the second rule with attribute phone as key.

On Table A.13 we have the results of applying the Adaptive SNJ for the first rule with phone as key. On Table A.14 we have the results of applying the Adaptive SNJ for the second rule with phone as key.
For this key value we did not use Metaphone as a similarity measure, since the key is only composed
with numbers. For the first rule with the attribute phone as key, the algorithm found every pair of records
as the Cartesian Product. However, for the same rule but with the attribute name as key, the algorithm
did not find every pair because the similar records were not adjacent and the similarity measure for their
keys was not high enough. Because of this, different windows were created and consequently, those
records fell into different windows.

<table>
<thead>
<tr>
<th>Measure 1</th>
<th>Measure 2</th>
<th>Measure 3</th>
<th>Measure 4</th>
<th>Measure 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold 1</td>
<td>77</td>
<td>77</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>Threshold 2</td>
<td>77</td>
<td>77</td>
<td>77</td>
<td>77</td>
</tr>
</tbody>
</table>

Table A.15: Results of the Adaptive SNJ tests for the first rule with attribute name as key.

<table>
<thead>
<tr>
<th>Measure 1</th>
<th>Measure 2</th>
<th>Measure 3</th>
<th>Measure 4</th>
<th>Measure 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold 1</td>
<td>51</td>
<td>50</td>
<td>51</td>
<td>63</td>
</tr>
<tr>
<td>Threshold 2</td>
<td>54</td>
<td>50</td>
<td>55</td>
<td>65</td>
</tr>
</tbody>
</table>

Table A.16: Results of the Adaptive SNJ tests for the second rule with attribute name as key.

On Table A.15 we have the results of applying the Adaptive SNJ for the first rule with name as key.
On Table A.16 we have the results of applying the Adaptive SNJ for the second rule with name as key.
In the second rule the values depended highly on the similarity measure that was chosen. As we can
see, for the attribute name we had better results when using Jaro-Winkler or Metaphone. In fact, as we
can see in Table A.16 for the second rule we can find all pairs as the Cartesian Product.

<table>
<thead>
<tr>
<th>Key 1 &amp; Measure 1</th>
<th>Key 1 &amp; Measure 2</th>
<th>Key 2 &amp; Measure 1</th>
<th>Key 2 &amp; Measure 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thresholds 1</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Thresholds 2</td>
<td>80</td>
<td>80</td>
<td>78</td>
</tr>
</tbody>
</table>

Table A.17: Results of the Canopy tests for the first rule.

<table>
<thead>
<tr>
<th>Key 1 &amp; Measure 1</th>
<th>Key 1 &amp; Measure 2</th>
<th>Key 2 &amp; Measure 1</th>
<th>Key 2 &amp; Measure 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thresholds 1</td>
<td>63</td>
<td>62</td>
<td>54</td>
</tr>
<tr>
<td>Thresholds 2</td>
<td>82</td>
<td>62</td>
<td>51</td>
</tr>
</tbody>
</table>

Table A.18: Results of the Canopy tests for the second rule.

On Table A.17 and Table A.18 we have the results of applying the Canopy for the first and second
rule respectively. As we can see, the tighter the interval of the thresholds the less pairs the algorithm
discovers. The effectiveness this algorithm also depends highly on the similarity measure that is chosen.
As we can see in Table A.18 for the attribute name as key, tight threshold 0.85 and loose threshold 0.65
we did found the 80 pairs of records. However, if the range of the threshold is smaller (i.e., tight threshold
0.9 and loose threshold 0.8), then some of the record pairs will not be found.
Appendix B

Created or Modified Classes/Packages

B.1 Created packages

The package src/functions was divided into the following packages:

- src/functions/stringmatching/generic - In this package we have the generic string matching functions (e.g., Jaro-Winkler, Levenshtein, etc).

- src/functions/stringmatching/domainspecific - In this package we have the string matching functions that are specific for each dataset.

- src/functions/stringmatching/deprecated - In this package we have older versions of string matching functions that are not being used.

- src/functions/standardize/generic - In this package we have the generic standardize functions.

- src/functions/standardize/domainspecific - In this package we have the standardize functions that are specific for each dataset.

- src/functions/standardize/deprecated - In this package we have older versions of standardize functions that are not being used.

- src/functions/merge/generic - In this package we have the generic merge functions.

- src/functions/merge/domainspecific - In this package we have the merge functions that are specific for each dataset.

- src/functions/merge/deprecated - In this package we have older versions of merge functions that are not being used.

- src/functions/generateid/generic - In this package we have the generic functions to generate IDs.

- src/functions/generateid/domainspecific - In this package we have the functions to generate IDs that are specific for each dataset.
- *src/functions/extract/domainspecific* - In this package we have the extract functions that are specific for each dataset.

- *src/functions/exceptions/domainspecific* - In this package we have the exceptions that are specific for each dataset.

- *src/functions/exceptions/deprecated* - In this package we have the exceptions that are not being used.

- *src/functions/datastructures/domainspecific* - In this package we have the data structures specific for each dataset.

- *src/functions/clustering/generic* - In this package we have the generic clustering functions (e.g., Transitive Closure).

- *src/functions/clustering/deprecated* - In this package we have old versions of clustering functions.

In the package *src/ajax/optimizer* the following packages were created:

- *src/ajax/optimizer/matching/structures* - In this package we have the structures that can be used by all algorithms to scale-up data matching.

- *src/ajax/optimizer/matching/similaritymeasures* - In this package we have all the string matching algorithms that can be used as similarity measures by the Adaptive Sorted-Neighborhood Join (SNJ) and by the Canopy algorithm.

- *src/ajax/optimizer/matching/scaleupalgorithms* - In this package we have all the classes that implement the algorithms to scale-up data matching.

- *src/ajax/optimizer/matching/scaleupalgorithms/traditionalblocking* - In this package we all the utility classes that are used by the Traditional Blocking algorithm.

- *src/ajax/optimizer/matching/scaleupalgorithms/sortedneighborhoodjoin* - In this package we all the utility classes that are used by the Traditional SNJ algorithm.

- *src/ajax/optimizer/matching/scaleupalgorithms/snjinvertedindex* - In this package we all the utility classes that are used by the Inverted Index SNJ algorithm.

- *src/ajax/optimizer/matching/scaleupalgorithms/canopy* - In this package we all the utility classes that are used by the Canopy algorithm.

- *src/ajax/optimizer/matching/scaleupalgorithms/adaptiveSNJ* - In this package we all the utility classes that are used by the Adaptive SNJ algorithm.
B.2 Created classes

In the package `src/functions/stringmatching/generic` and `src/ajax/optimizer/matching/similaritymeasures` we created the following classes:

- `Jaccard.java` - This class implements the Jaccard similarity coefficient.
- `JaroWinkler.java` - This class implements the Jaro-Winkler algorithm.
- `Levenshtein.java` - This class implements the Levenshtein algorithm.
- `Metaphone.java` - This class implements the Metaphone algorithm.
- `Soundex.java` - This class implements the Soundex algorithm.
- `TFIDF.java` - This class implements the Term Frequency/Inverse Document Frequency (TFIDF) algorithm.

In the package `src/ajax/optimizer/matching/structures` we created the following classes:

- `ComposedKey.java` - This class contains the constructor of a composed key.
- `Item.java` - This class is the structure used by the algorithms to scale-up data matching.
- `Record.java` - This structure contains all the fields and values of a given record.

In the package `src/ajax/optimizer/matching/scaleupalgorithms` we created the following classes:

- `AdaptiveSNJ.java` - This class has the implementation of the Adaptive Sorted-Neighborhood Join (SNJ) algorithm.
- `Canopy.java` - This class has the implementation of the Canopy algorithm.
- `MultiPass.java` - This class has the implementation of the Multi-pass algorithm.
- `SNJInvertedIndex.java` - This class has the implementation of the Inverted Index SNJ.
- `SortedNeighborhoodJoin.java` - This class has the implementation of the Traditional SNJ.
- `TraditionalBlocking.java` - This class has the implementation of the Traditional Blocking.

In the package `src/ajax/optimizer/matching/scaleupalgorithms/sortedneighborhoodjoin` and `src/ajax/optimizer/matching/scaleupalgorithms/adaptivesnj` we created the class `SlidingWindow.java` (with different implementations in each package). This class corresponds to the sliding window used in both Traditional SNJ and Adaptive SNJ algorithms.

In the package `src/ajax/optimizer/matching/scaleupalgorithms/canopy` we created the following classes:

- `Cluster.java` - This class corresponds to a cluster where several records will be inserted into. The use of clusters is part of the Canopy algorithm.
• **TokenGenerator.java** - This class generates the tokens used during the *Canopy* algorithm.

In the package `src/ajax/optimizer/matching/scaleupalgorithms/invertedindexsnj` we created the following classes:

• **InvertedIndexItem.java** - This class corresponds to an item that is inserted in the inverted index table. This inverted index table is used in the *Inverted Index SNJ* algorithm.

• **SlidingWindow.java** - This class corresponds to the sliding window that is used in the *Inverted Index SNJ* algorithm.

In the package `src/ajax/optimizer/matching/scaleupalgorithms/traditionalblocking` we created the class **Blocking.java** that corresponds to the blocks used in the *Traditional Blocking* algorithm.

In the package `src/ajax/expression` we created the class **Hint.java** that corresponds to a hint. This hint contains a type and a value.

### B.3 Modified classes

The following classes were modified:

• **bin/ajax/parser/CleaningParser.jj** - This file was modified in order to parse the hints that are specified in the data cleaning program.

• **src/ajax/catalog/Matching.java** - This class was modified in order to write the Java code according to the chosen algorithm to scale-up data matching and corresponding parameters.

• **src/ajax/catalog/Function.java** - This class was modified in order to create stored functions into the Relational Database Management System (RDBMS).

• **src/ajax/type/ComposedType.java** - This class was modified in order to create the composed types in the RDBMS. This composed types are used/returned by the stored functions.

• **src/ajax/catalog/Table.java** and **src/ajax/db/Relation.java** - These classes were modified in order to support queries that are specific for the *PostgreSQL* RDBMS and the *Oracle* RDBMS.

• **src/ajax/catalog/Transformation.java** - In this class we added methods that set the algorithm to scale-up *data matching* that is going to implement a given transformation. This choice is given by the Optimizer.

• **src/ajax/expression/Expression.java** - This class was modified in order to generate the code according to the chosen algorithm to scale-up *data matching*.

• **src/ajax/optimizer/Optimizer.java** - In this class we added support for the Optimizer to access the hints provided by the user in order to set for each transformation its physical implementation. This decision is made based on the hints that the user wrote.