**Universidade de Aveiro** Departamento de Eletrónica, Telecomunicações e Informática
**2013**

**FREDERICO SANTOS HONÓRIO**

# Plataforma de Mapeamento de Registos Multi-Domínio

# Multi-Domain Record Linkage Platform

**Universidade de Aveiro**
**2012**

Departamento de Eletrónica, Telecomunicações e Informática

**FREDERICO SANTOS HONÓRIO**

# Plataforma de Mapeamento de Registos Multi-Domínio

# Multi-Domain Record Linkage Platform

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. Carlos Costa, Professor Assistente do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

**o júri / the jury**

|  |  |
|---|---|
| presidente / president | Prof. Doutor António Manuel Melo de Sousa Pereira<br>Professor Catedrático, Universidade de Aveiro |
| vogais / examiners committee | Prof. Doutor Rui Pedro Sanches de Castro Lopes<br>Professor Coordenador, Dep. Informática e Comunicações da Estg do Instituto Politécnico de Bragança |
|  | Prof. Doutor Carlos Manuel Azevedo Costa<br>Professor Auxiliar, Universidade de Aveiro |

**agradecimentos / acknowledgements**

Agradeço o trabalho de André Ventura, Filipe Manco, José Mendes e Nuno Silva.

Agradeço também ao meu orientador Carlos Costa, e ao meu colaborador, Luís Ribeiro, pela ajuda e paciência oferecidas.

**palavras-chave** mapeamento de registos, resolução de identidade

**resumo** Esta dissertação propõe e apresenta uma ferramenta tecnológica que permite realizar mapeamento de registos. A abordagem proposta começa por definir uma sequência de tarefas necessárias para efetuar o mapeamento de registos. São depois discutidos métodos de separar estas tarefas em unidades de trabalho reduzidas. A solução baseia-se numa arquitetura composta por vários executores que levam a cabo essas unidades de trabalho de uma forma paralela, objectivando-se um processo mais rápido. As vantagens da utilização desta abordagem em diferentes contextos são também estudadas.

**keywords** record linkage, identity resolution

**abstract** This dissertation proposes and presents a technologic framework to perform record linkage. The proposed approach begins by defining a sequence of tasks necessary for record linkage. Then, several methods to split these tasks in small work units are discussed. The solution architecture is based on various executors that carry out the work units in a parallel manner, thereby making the process quicker. Finally, the benefits of using this approach in different contexts are also presented.

# Contents

# List of figures

# List of tables

# Acronyms

**CSV** — Comma-Separated Values

**EMPI** — Enterprise Master Patient Index

**ETL** — Extract, Load, Transform

**SQL** — Structured Query Language

**SaaS** — Software as a Service

**XML** — Extensible Markup Language

**SOAP** — Simple Object Access Protocol

**SDK** — Software Development Kit

**NLP** — Natural Language Processing

**PIX** — Patient Identifier Cross-referencing

**XDS-I** — Cross-enterprise Document Sharing for Imaging

# Chapter 1 - Introduction

Over the last two decades there was a growth of information systems within several sectors of human activity (e.g. industry, health[1], finance). Information systems are developed to support or improve the access, storage or distribution of information. Typically the information systems are tailored to one domain following an ad-hoc approach. The problem appears when information from several sources needs to be combined into an integrated cross-domain information system. Since every source follows a different modelling approach direct linkage of records is in most cases not possible. Frequently, the records to be matched correspond to entities that refer to people (e.g. clients, patients, students, employees) or to people's documents (e.g. electronic health record, images, articles). Another issue is the possibility of input error or data corruption, adding another level of complexity to the task.

The task of record linkage is now commonly used for improving data quality and integrity, to allow re-use of existing data sources for new studies, and to reduce costs and efforts in data acquisition[2].

These techniques have a great impact in health-related information systems, where there is a common need to combine information from several registries. Other fields, such as statistics, data integration or data warehousing can also benefit from these techniques.

Due to the heterogeneous nature of the problem, record linkage solutions tend to be highly specialized (i.e. developed just for solving one specific problem). Transparent re-use of the record linkage routines is not a realistic scenario. The record linkage workflow may have well defined stages to extract, clean, transform and, export data [3] however each stage requires a lot of re-configuration to deal with new problems. Therefore, many organizations turn to developing applications in-house from scratch. Thus, an ideal solution would be a linkage tool that is flexible and generic enough to adapt to new scenarios and, at the same time, may be reassembled in a convenient manner.

The need for a generic record linkage tool can in part be shown by some recent examples ([4][5][6]). Certainly the specific domains may differ, and the analysis goes beyond the simple application of record linkage to the acquired data. However, there are common procedures that tend to be developed independently even though they are very similar.

It is seen that such a tool is useful for such different areas of study or domains because it reduces the effort of developing a record linkage solution to the configuration and implementation of functionality specific to the problem.

We propose a plugin framework where registered users may upload their plugins within the record linkage workflow. The plugins type may vary from record linkage algorithms, to import/export data interfaces or data cleaning algorithms according to the several stages of the record linkage workflow. Such a tool, while still requiring programming knowledge, would severely reduce the effort necessary to develop a record linkage solution.

In this chapter, we introduced the concept of record linkage and the impact it can have in several fields of study. The following chapter presents a study of the field of record linkage, from a generic perspective, evaluating the current methods and approaches and establishing the requirements for a competing record linkage tool. This chapter also includes a survey of several tools that can be used for record linkage purposes. Chapter 3 identifies the requirements of a record linkage tool that is generic and efficient. Chapter 4 will introduce our proposal for a framework that fits the established requirements, also mentioning the paradigms techniques and tools used. Chapter 5 will include an evaluation of the work with case studies and results, and discusses the merits of the work performed. In the sixth chapter we discuss the contributions made to the field and look at future work to better to enhance this work.

# Chapter 2 - Background

This section serves as a survey the state of the art in the topic of record linkage. It aims to identify the current technologies and methodologies, as well as the gaps and possible improvements to the field. We now discuss the general approach to record linkage, identifying the common steps and techniques involved.

## *2.1 - Approach*

Record linkage operations deal with records, a structure composed of fields which contain information on a particular entity. These records can be stored in distinct data sources, and there can be a necessity to collate such information. This format of data is the focus of record linkage techniques.

### 2.1.1 - Deterministic Record Linkage

When dealing with data with unique identifiers, a process of deterministic record linkage can be employed [7]. Deterministic record linkage is the process of matching the unique identifiers of the records in each data source directly, and linking records which have the same identifier. There are several problems with maintaining unique identifiers, however. Many systems are not designed to share unique identifiers, causing such values to be unique only within the system. The task of record linkage can be employed when the records of a small system needs to be integrated into a larger system, a case that is not often anticipated by the designers of the first system. In this case, even when several smaller systems store similar, homogenous data, the aggregation of such data can not be made by their unique identifiers because the distinct systems have divergent identifiers. Even when such identifiers are shared, larger data systems are prone to input errors, data loss and corruption.

### 2.1.2 - Probabilistic Record Linkage

Given the limitations of the deterministic method we now outline the process of probabilistic record linkage. Fellegi and Sunter [8] described a probabilistic model to recognize matching records, and defined a matching function as a function that operates on a pair of records and assigns it to one of three categories:

- Positive link - when records belong to the same entity.
- Positive non-link - when records belong to different entities.
- Possible link - when there is not enough confidence to affirm any of the two.

As mentioned, the existence and accuracy of a unique identifier for each record is not presumed. Instead, a set of fields that describe the entity sufficiently must be chosen. These fields can be called quasi-identifiers [9]. It is apparent that a single or too few quasi-identifiers are not sufficient to identify an entity, so the choice of quasi-identifiers is important to the process. It is also important that the data for the chosen quasi-identifiers has a high degree of correctness. For example, while a person would be correctly identified by the social security number, it could also be so by the full name, date of birth and area of residence. If any of these fields had wrong or missing values, a different, ideally larger, set of fields would be necessary to assure higher accuracy.

**Data Cleaning**

An important step to prepare for the record linkage process is the standardization, or cleaning, of the compared data. Not only must the records match structurally, by having the same set of fields with the same data type each, but also lexicographically. This means the formats and constraints for each field must be the same, to avoid false-negatives. An example would be two data sources where the first has all personal names capitalized, and the other does not. Parsing, if necessary, also belongs in this step. Data cleaning also includes normalization of synonyms, abbreviations and acronyms of the data. This ensures consistency for all the data, which is important when exact matching is not possible. An example of this step is shown in Table 2-1.

| Field | Original values |
|---|---|
| Name | Frederico S. Honório |
| Address | R. de Fora, 9990-303 Alcobaça |

| Field | Values after cleaning |
|---|---|
| First Name | Frederico |
| Last Name | Honório |
| Street Addr. | R. de Fora |
| Postal Code | 9990-303 |
| City | Alcobaça |

Table 2-1 Example of a data cleaning process.

Methods for achieving probabilistic standardization of names and addresses have also been developed [10] by using hidden Markov models [11].

**Probabilistic Matching**

Once the records are homogenized, matching fields is then a matter of using a function that calculates the similitude of the values for the fields in each record and produces a score for the pair of fields. This function is dependent not only on the type of information (numbers, dates, strings of characters, etc.) but also its meaning (addresses, postal codes, names, etc.). The function should produce the highest score when the values of the fields are equal and a progressively lower score as the differences between the values increase. Some methods of matching field values are now illustrated.

There are several methods of comparing strings of characters, one of them being the Levenshtein distance [12]. This method calculates the number of character insertions, deletions and substitutions required to transform a string into another. A value of zero indicates the strings are equal and this value increases as the strings differ. This value needs to be normalized to be used as a score. Table 2-2 presents some examples:

| Word 1 | Word 2 | Distance | Explanation |
|---|---|---|---|
| alchemy | alchemist | 3 | 1 substitution and 2 insertions |
| wispy | wisp | 1 | 1 deletion |
| reverse | reave | 4 | 1 insertion and 3 deletions |

Table 2-2 Example of Levenshtein scores.

A similar method is the Jaro-Winkler distance metric [13], which produces a score between 0 and 1 based on matching characters and transpositions, with 1 being an exact match and 0 is a completely distinct string.

Phonetic algorithms are also used to compare strings, however, these usually deal with names. Given the evolving nature of language, often a spoken name can be written in several manners. This is a problem when said names are conveyed by speech and then written, leading to inaccurate spellings of a name. Such problems can impact record linkage processes if names are involved. To tackle such problems, several phonetic algorithms have been developed. A phonetic algorithm is able to map words that have the same pronunciation but different spelling to a common symbol, allowing multiple spellings of a name to be represented in the same manner. Such algorithms are usually designed for a specific language, like, for instance, Soundex, that was designed for English.

We have given some examples of matching strings, but there are cases where these alone are not sufficient. Consider the differences between matching the names of two people and matching two addresses. In this case, even though both fields are character strings, it might not be sufficient to use string similarity metrics like the described previously. An algorithm that can take into account common address abbreviations would better compare the addresses. On the other hand, personal names can benefit from using a phonetic algorithm to reduce spelling inaccuracies.

With a means to match field values, the process of matching records can now proceed. Each field is then attributed a weight according to their significance in the resolution of the identity. The Fellegi-Sunter algorithm describes an accurate mathematical method of assigning these weights [8]. A method to apply the EM (expectation-maximization) algorithm [14] in the computation of field weights has also been described [15].

The final matching of records is then made by deriving a score for the pair. The score is the weighted sum of the scores for each of the fields in the two records, using the weights defined for each field. Finally thresholds that ascribe the pair to one of the categories are defined. Two thresholds exist, the high and low thresholds (Figure 2-1). When the score is above the high threshold, the pair is classified as positively linked, when it is below the low threshold, it is classified as positively non-linked, and when it is between the high and low thresholds it is possibly linked. The possibly linked records are eligible for clerical review that can be a finite resource and the definition of the thresholds should take that into account.

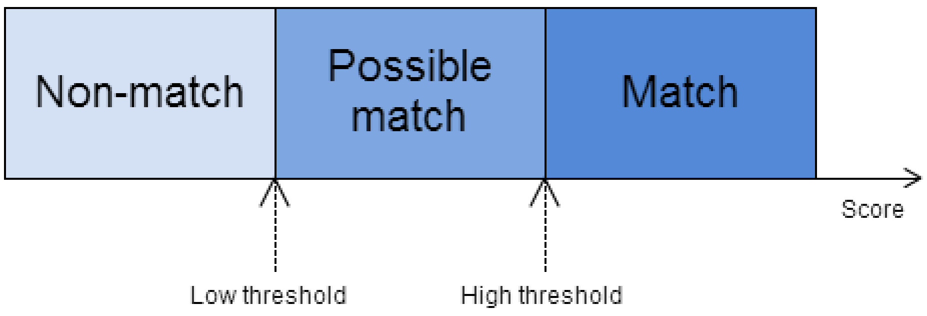

Figure 2-1 - Illustration of the thresholds.

Given both the definitions of probabilistic and deterministic record linkage, we can describe the deterministic method as a special case of the probabilistic one. Particularly, the deterministic option does not allow for possibly-linked records, which, in the probabilistic method, translates to having equal high and low thresholds, therefore allowing only matches are non-matches. Moreover, the deterministic method uses only unique identifiers to match records, which is simply done by ascribing the maximum weight to the unique identifier, ignoring the remaining fields. Finally, the deterministic method uses exact matching to compare the unique identifiers, meaning the method of comparison will not use a continuous score (that would denote various degrees of similarity), but instead use only the maximum and minimum scores, to represent a match or a non-match. With this combination of rules, only the records where the unique identifier is equal are linked.

### 2.1.3 - Blocking

The naïve approach to both deterministic and probabilistic record linkage would require the comparison of every record against all the records besides itself. With $N$ records from a single source, this would result in $O(N^2)$ order of comparisons, and with two sources with $N$ and $M$ records, $O(N\ M)$ comparisons.

The need for accepting new records and performing only the necessary comparisons becomes evident, as only $N$ or $M$ comparisons are necessary depending on the set the new record belongs to. Without a means to add new records dynamically, a full comparison would be necessary.

With this naïve approach, the cost gets increasingly prohibitive has the number of records grows, but there are some methods to improve this.

A blocking method [16] creates groups, or blocks, of records according to some established criteria that uses the values from the record fields. For example, a blocking technique might separate the records with personal information by the gender or the year of birth. Each record is then only matched against the remaining records in its block. This means the number of pairs, and therefore comparisons, is reduced substantially. Depending on the data set, the assumption that values of a blocking field are always correct cannot be done. This is a problem because it is necessary that the blocking function does not separate records that would otherwise have a positive link as it would reduce linkage accuracy.

Another method to reduce the complexity of comparing records is the Sorted Neighborhood method [17] (Figure 2-2). This method relies on creating, for each record, a key from the values, and then sorting the records by using this key. Then, a window is established that defines the number of records in a record’s vicinity. Every record is then compared with the records in its vicinity. An example of this is using the date of birth to compare personal records and matching only records where the date of birth is within a year of the selected record.

The success of this approach is extremely dependent on the method for computing the key. In fact, it is often necessary to execute multiple runs of this method, with distinct keys, for two reasons:

- It is possible that the values used to create the key are incorrect.
- It is difficult to construct a key that factors values from multiple fields while maintaining a small window. Increasing the window to large amounts reduces the benefit from this technique by increasing the number of comparisons to be made for each record.

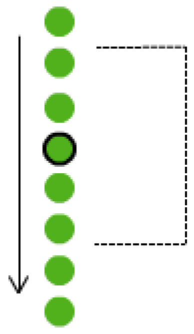

Figure 2-2 - Illustration of the sorted neighbourhood method. The disks are the records, the highlighted disk is the record being compared, the dashed line shows the window of records to match against.

A similar technique, that supports record linkage in a large data set has been achieved by using a set of pre-processing methods and a method of sorting [18].

The use of canopies [19] (Figure 2-3), a looser method of blocking, can also be used to reduce the total number of comparisons made. This method places records into clusters through a computationally inexpensive metric. Moreover, it allows clusters to overlap, allowing the same record to be placed in multiple clusters. The records in the clusters are then matched pairwise. This approach is safer because records can be in multiple clusters, and so having incorrect values is not as harmful as the previous methods. However, the choice of the metric is still very important, as possibly linked records may not fall in the same canopies.

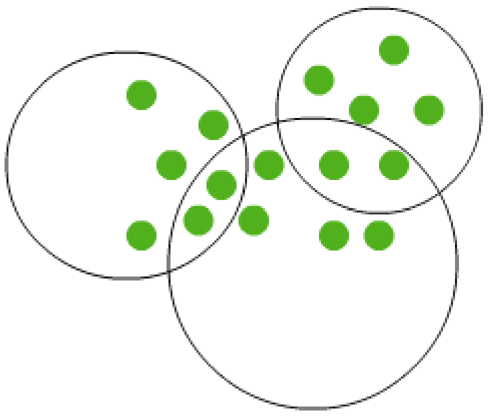

Figure 2-3 - Illustration of the canopy method. The disks are records, the circles are the canopies.

These previous techniques are also useful for parallelization, and there are several other methods for this. One possible approach is utilizing a blocking technique and assigning the separate blocks to separate processors. This of course relies on the previously stated assumptions about the correctness of the values in the fields used by the blocking methods. As there is no guarantee that the blocks have similar sizes, the workloads for each processor may very significantly. The same holds true for using canopies to distribute the record linkage process.

P-Swoosh [20] addresses this in two possible ways. The first, by sharing the complete record sets between all processors, and determining which pairs of records each processor analyses. This is based on the number of processors available to assure the workload is evenly distributed. The second, by dividing the record set into buckets and sharing a reduced number of buckets between processors. This reduces the communication and storage costs per processor, but adds a layer of synchronization.

### 2.1.4 - Private Record Linkage

In some applications of record linkage, especially when personal data is analysed, there can be a concern about the privacy of the process, and methods of private record linkage can be used. An approach where two parties that own private records can perform record linkage without plain-text record exchange has been defined [21]. Records are encoded as complex numbers, based on previously agreed upon values, and are then exchanged between the two parties involved, which use them to compare against their records. Another method [22] uses several hashing and encoding methods on a bloom filter derived from the private data, achieving private record linkage. It is important to notice that both of these methods require some form of agreement between the two parties.

A different method to have private record linkage is the use of homomorphic encryption [23]. Homomorphic encryption is a type of encryption where the ciphertext can have some operations performed on it without being deciphered, and once it is deciphered the plain text maintains the result of the operations. As an example, with the texts $a$ and $b$:

$$c_a = Encrypt(a)$$
$$c_b = Encrypt(b)$$
$$R = Concat(c_a, c_b)$$
$$S = Decrypt\ (R)$$

results in $S$ having the value of the concatenation of $a$ and $b$. In this case, it is said the encryption algorithm is homomorphic regarding concatenation. An encryption scheme is said to be fully homomorphic if it allows a functionally complete set of binary operations. Functional completeness is a property of a set of binary operations which can express any truth table by combining the operations in a logical expression. Both the NAND and NOR operations are functionally complete as is the set of AND and XOR. We can then express any complex binary function with a Boolean expression consisting of only these operations. The importance of this property is that a fully homomorphic encryption scheme allows the operations to be performed without explicitly knowing the value of the plaintext, resulting in the possibility of offloading such calculations to a third party, privately. In the advent of cloud computing, delegating complex processing tasks to powerful distributed systems is a usual occurrence, being able to do so privately is a significant change in the way we can use such services. The drawback of this method is the performance of the operations performed. Complex routines must be converted to logic networks, a step which is cumbersome and non-trivial. However, recent improvements on these techniques begin to make the use of this form of encryption a real world scenario. Given the outlined approach to record linkage, it is apparent that these methods could be of use in the comparison step. With the use of homomorphic encryption, methods of comparing ciphered fields could then produce a score for each field and allow the records to be classified in respect to their similitude, without knowing the plain text values of the fields.

### 2.1.5 - Merging

A possible final step in record linkage is merging the linked records. Once the records are linked, merging a pair of records is the problem of generating a new record with the most complete information, derived by the two records. Table 2-3 illustrates this step:

| Field | Record 1 | Record 2 |
|---|---|---|
| Name | Fred S. Honório | Frederico Honório |
| Address | Rua de Fora | R. de Fora, Alcobaça |
| Phone Number | 777123123 | - |

Table 2-3 Example of linked records.

There are several issues with presented example. Firstly, none of the records have the most complete name: Record 1 has an abbreviation of the first name and has the initial of the middle name and Record 2 has the complete first name, but no middle name. The most complete form of the name would be “Frederico S. Honório”. The address field has a similar example, where “Rua” (street) is abbreviated has “R.”, in this case however, it could be beneficial to keep the shorter form as long as all the merged records conform to such rule. The phone number presents no issue, the phone number is only included in one of the records, so there is no conflict. The resulting record in this case would be:

| Field | Record 1 + 2 |
|---|---|
| Name | Frederico S. Honório |
| Address | Rua de Fora, Alcobaça |
| Phone Number | 777123123 |

Table 2-4 The merged record.

We can see that this process is not trivial and is highly dependent on the type of data that is being linked. It is difficult to anticipate the completeness of the information in each record, which makes the task complex, and often left to be solved by clerical review.

## *2.2 - Software Tools*

Now that we established the essential processes of a record linkage application, we can analyze the current tools that propose to solve these problems. Ahead are discussed several software applications that tackle the problem of record linkage.

### 2.2.1 - SERF

The SERF [24] project provides a Java framework for generic record linkage, offering a set of data structures and algorithms commonly used in record linkage tasks.

The framework has three important components, the Record data type, the MatcherMerger interface, and an iteration algorithm. In solving a particular record linkage problem, a developer would need to transform his data in a list of Record objects, and create or choose an already provided class implementing the MatcherMerger interface. This interface imposes a method for matching two records and merging them. The matching method has a Boolean result, meaning that it is impossible to extract groups of possibly linked records, only of positive links and positive non-links.

The MatcherMerger implementation can then be used by an algorithm that iterates through the records and applies the matching and merging methods. Some of the SERF authors propose a number of algorithms for iteration [25], but not all of these are implemented in the framework.

This work adds important contributions to the field, the most important being the Swoosh family of algorithms. However, using this framework directly has a few problems: the absence of a possibly linked category means that the matching function needs to be very accurate, for fear of discarding pairs that would be eligible for manual inspection. Another problem is that the matching and merging processes are strongly coupled, which means that benchmarking various matching algorithms or thresholds for a specific problem will take more developing effort.

### 2.2.2 - Match2Lists

Match2Lists [26] is a commercial web solution that implements record linkage on text lists in CSV (comma-separated values) format or similar.

Lists are first uploaded and then described, assigning a data type (name, email address, etc.) to the columns of the list. There are three relevant operations available: matching two lists, merging two lists and removing duplicates on a single list. All these operations require several matching functions, which are inferred by the data types defined in each list. These types however are very restrictive as there is no option to expand them, one example being the lack of dates as a data type.

The lists are then automatically processed and the fields with the same data type are compared across all records, producing a score for each field in every pair of records. These scores are combined in a final score for the pair, which represents the likelihood of records corresponding to the same entity. The method of determining this score is not fully disclosed however, and assigning weights for each field would allow a more precise score by defining which fields are more likely to relate distinct records.

Then, a user interface is presented to allow manual approval or rejection of matches and defining the thresholds for automatic approval and rejection. This process is very intuitive, showing records side by side with all the fields represented. The result is also a CSV file containing a list of matching records, a list resulting from the combination of two lists or a list without duplicate records depending on the chosen operation.

In summation, while this tool presents a useful interface for analysis of linkage results, it lacks flexibility in score calculation and methods of inserting data and exposing the results.

### 2.2.3 - Open EMPI

Open EMPI [27] proposes to solve the problem of unifying patient health records given that this data comes from dissimilar systems. The system provides methods to clean and standardize input data and a series of blocking and matching algorithms. As a software framework, it allows the matching methods to be extended, providing added flexibility. The system designed in a data integration perspective, allowing queries to be performed on the collected data.

From a record linkage perspective, while very flexible, this work lacks generality as it was designed for medical data exclusively.

### 2.2.4 - Duke

The duke engine [28] uses a novel approach. Instead of using files or databases, the records are imported into a Lucene instance. Lucene is a text search engine that allows fast indexing and search.  This means that the records are not directly compared pair-wise, as detailed before. Instead, the values of each field are tokenized and then, a search is performed for each token, and the results of this search are the records eligible for comparison. The records are then compared by a matching function. The greater flaw in this approach is that it requires an exactly matching token, or a token within a certain edit-distance, as those are the methods for matching tokens available to Lucene.  As an example, consider two records describing the same city by its name and population: (Lisbon, 547631) and (Lisboa, 550000). Assuming the existence of matching functions that would classify these records as positively linked, they would not even be selected for comparison because no tokens match, and therefore would not be classified as positively linked. The project's architecture allows extensions, namely of the importing and data cleaning functions as well has field comparators. This makes a very flexible tool, which is also reported as very performant.

This tool is very powerful and has a good degree of flexibility, but it is severely dependent on the search methods provided by Lucene, which limit its accuracy and generality.

### 2.2.5 - Kettle

Kettle [29], also known as Pentaho Data Integration, is an ETL (Extract, Transform, Load) tool. ETL refers to a set of processes commonly used in data warehousing to move data between databases, performing the necessary data transformations in between. Because record linkage problems can deal with large sources of data (such as databases), and a de-duplication process is often necessary, this tool showed potential in dealing with record linkage operations.

Kettle is an open-source project that allows the design of an ETL workflow, by utilizing a series of plugins to introduce data into the system (extract), modify it (transform), and exporting it to other systems (load). These plugins are called Step Plugins, and are mainly row-oriented, in the sense that each plugin accepts and produces a list of rows. A plugin can not only modify the individual fields, but also alter the row's structure, produce values dynamically or omit a row altogether, among others. The steps are then chained together in a graphical representation, allowing the user to verify and test each step in the process. Each step can have a configuration, which is defined in a user interface window. Kettle provides plugins for accessing data from a multitude of sources, from CSV files to SQL and NoSQL databases. Also available are plugins for field validation and cleaning, row sorting and other operations. In order to express simple logic, a user can create Java or Javascript code and use it with the respective plugin to process the rows of a step. It is also possible to create a plugin, also in Java, that can use libraries and more complex functionality. Each plugin works asynchronously, in a streaming fashion, accepting a single row as input as soon as possible and outputting the transformed row once available. This quality is useful to reduce the overall runtime of the complete transformation, as this allows the individual steps to run in parallel as long as there are rows available as input. This set of connected steps is called a Transformation.

Also available are Job Entry Plugins that produce a Boolean value by verifying a condition necessary for the following job, and only execute if the preceding value is true. Both Job Entries and Transformations can be chained, effectively providing flow control over the ETL process. A series of connected Job Entries and Transformations is a Job.

The project also allows for remote execution of Jobs and Transformations using the Carte server. It can be used by designing the Transformation in Kettle and choosing to run the transformation in a Carte remote instance.

The application of record linkage with this tool is not readily available. The Fuzzy Match plugin has several string matching algorithms available and can output the score given to a match. This however assumes a reference set of rows, which are kept in memory and as a result cannot be very large. The plugin works by preloading the available rows from the reference plugin, and matching each incoming row from the other sources with the references. Also, this plugin only works with strings, so fields with other data types could not be used to evaluate similarity.

# Chapter 3 - Requirements

The goal of this dissertation is to develop a software framework that could be used to solve generic record linkage problems by allowing users to configure it and provide code in order to adapt the provided functionality to specific problems.

In this section, the requirements for such a tool will be presented.

## *3.1 - Functional requirements*

We now describe the required functionality for this system while taking into account the studied background.

### 3.1.1 - Interaction with external data sources

The process of record linkage deals with the manipulation of records to create a mapping that associates said records when they refer to the same entity. For this to happen in the framework, it is necessary to extract such data from the source and transform it into a format that could be handled. This task is made difficult because the data source is not known beforehand, which means that neither the method of storing the data is known, nor is the data schema.

### 3.1.2 - Allowing dynamic sources of data

In addition to being able to fetch data from a source, it is also important to have a signalling mechanism to alert the framework when new data is made available. In this case, a user (or system acting as one) can introduce new records into a set of data that has already been extracted, updating it. This implies that the framework can be used not only for a static source of data, but also in a dynamic process of data production.

### 3.1.3 - Record Matching

The process of matching the records is a core issue. The framework must allow records to be matched, maintaining the idea that no information about the data is known beforehand. The matching process must apply the method of probabilistic record linkage, previously described, and the framework must allow the user to define the weights and thresholds necessary.

### 3.1.4 - Exporting data

The system should allow us to extract the matching information out of the system, in a relevant manner. Again, there can be no assumptions made about the data that has been matched, or the type of result produced in the matching process. A user might require a simple list of the matching records, but can also need that all the records that match the same entity are merged into a new record with the most complete information.

### 3.1.5 - Use of Plugins

To enable the use of customized and complex routines, designed by users, the framework must support a plugin mechanism. This means that the user should be able to upload plugins to the framework. The framework should then be able to load the plugin into memory and be able to execute it, thereby integrating it into its routines.

### 3.1.6 - Benchmark-oriented approach

The process of record linkage is often iterative. The several string matching methods, exposed in the background chapter, are an example of the multiple approaches that can be followed in the matching process. Therefore, testing different approaches with minimal effort is important to the quality for this type of tool. As such, the user must be able to reuse plugins and data in the framework. Once the data is brought into the framework, there is no need repeat the data extraction process from the source. The same applies for the data used in other steps. Being able to repeat a process easily is also important to assist in the development of plugins, as developers can replace new versions of the plugin and verify the results with less effort.

## *3.2 - Non-functional requirements*

With the explicit functionality described, we now specify the requirements to the operation and design of the system.

### 3.2.1 - Privacy and Confidentiality

Confidentiality is the act of controlling which entities access some private data. Confidentiality is an extension of privacy. Privacy is at the personal scope, in the sense that individual has the right or desire to control by whom data can be accessed. Due to the fact that the nature of the data is not known in advance it is possible that privacy rules are required, an example being when dealing with medical record linkage. Because the system has multiple users, it is necessary to restrict the access to data to only the appropriate users.

### 3.2.2 - Security

The framework will be able to execute code provided by an untrusted external source. This is an important security concern, because the external code can be harmful to the system and the data contained in it. Moreover, the framework must not allow that user provided programs diminish its capabilities significantly by abusing system resources like the processor or memory. Also, the user code should not be able to corrupt the data in the framework permanently or interfere with the users tasks.

### 3.2.3 - Scalability

Record linkage problems often deal with large volumes of data, from various sources. We have illustrated how such processes can be time-consuming because the naïve approach requires comparing all pairs of records. It is then advantageous to have a tool that is able to scale by leveraging the addition of new hardware.

Having a scalable tool allows for us to cope with larger problems by more efficiently using the hardware provided.

### 3.2.4 - Robustness

Another issue associated to user code is the effect such code can have on the robustness of a system. Errors during execution can halt the execution of the system, making it unreliable and unable to serve its purpose. By accepting foreign code, a system must be prepared for errors and exceptions generated by the code. Also, misuse of the framework by said code must not hinder execution. Because the system and the user code communicate directly, establishing a programming contract allows the system to enforce a certain pattern of use and actions performed by the user code can be verified to conform to said contract.

### 3.2.5 - Interoperability

Although record linkage can be useful by itself, it may be a single part of a more complex problem. As explained previously, integrating data from several systems is a possible use for this platform. However, it is also likely that such data is then placed in another system, making the platform a vehicle for this transaction. For this and other use cases it is important to allow the integration of the framework with a set of system. Even though the functionality of the framework should be exposed to users, it is apparent that allowing external systems to interact with the framework as a client is also advantageous.

# Chapter 4 - Framework Proposal

There are record linkage solutions that excel in several areas. However, it was identified the necessity of a tool that could implement all the important features described, in a flexible way. We propose a framework that leverages the techniques described while requiring only a small adjustment effort to a specific record linkage problem. Moreover, it was also identified the necessity of user code injection, providing this way a means to adjust the framework to a wide range of problems.

Developing the tool with a SaaS (Software as a Service) model has a greater potential in collaboration environments, which given the possible scale of record linkage problems is also desirable. In addition, it is also consistent with the principle of simplifying the resolution of record linkage problems.

Also important is the ability to add records to a data set dynamically if a given dataset can change. As stated before, it is more efficient to compare the new record with the existing records instead of repeating the whole procedure and also allows more varied scenarios of usage, where the data source is not static. Additionally, it is a requirement for interoperability with other systems, which also fits with the SaaS model.

The plugin-enabled architecture of Kettle is a good approach to achieve the desired generality. In fact, our first proposal — outlined in Appendix A — was based in the extension of Kettle to support our record linkage requirements. However, this approach has several drawbacks due to thestreaming nature of Kettle and the effort required to produce a useful solution.

Inspired on Kettle experiments, the proposed framework was designed to allow the creation of plugins to import, transform, compare, and export data or choose plugins already available. The platform would also expose a web service that could receive a new record, add it to the dataset, and perform the necessary comparisons to check for matches. Additionally, it could be used as a benchmarking tool, allowing users to test competing matching heuristics, field weights, and methods of merging matching records, for example.

We now outline a framework devised for this purpose of general record linkage, first describing a general record linkage workflow, and then describing the architecture to implement such solution.

## 4.1 - Framework data types

The interactions within the framework often require using several data structures. So, we decide to present some of these before the workflow to better understand their role. Moreover, these structures may also be populated by the user plugins.

We start by defining the Record structure that simply holds a list of field values, all of them sequences of characters (strings). An object of this type should refer to a single entity. To provide the data in a Record with semantic meaning (metadata), the Schema data type was created. The Schema defines the names of the fields in a Record, also as a list of strings. This implies that a Record is meaningless without the corresponding Schema, but also means that the metadata is not included in every Record. This is a more sensible approach, otherwise there would be a significant overhead in the process of communicating a set of Records, as the metadata for each record would need to be repeated each time a single record is transmitted, even if all Records in a set share the same metadata.

| Schema | Record |
|---|---|
| Order number | 1031354286 |
| Date | 2-1-2013 |
| Total | 35,5 |

Table 4-1 An example of a Record and the associated Schema.

The result of a successful match is expressed in the Result type. This type comprises a reference Record and a list of Matches. A Match is defined by the matching Record and the match score. Matching is always done comparing a reference Record against another Record. As such, if a match between the records $a$ and $b$ occurs, the Result $r_a$ will be composed of $a$, as a reference Record, and a Match $m_1$ with $b$, as a matching Record, and the comparison's score. If a $c$ Record also positively matches $a$, then a Match $m_2$ with $c$, as the matching Record, and the corresponding score is added to $r_a$. The reasoning behind the choices of these data types is described in section 4.4.

## *4.2 - Workflow*

Given the background study, a framework workflow was devised to support defined record linkage requirements. It is composed of various steps, described ahead.

### 4.2.1 - Import Step

This step deals with bringing the data into a form the framework can process. More specifically, the import step creates Record objects which can then be used by the framework. Access to resources outside the system and subsequent parsing is done at this step.

### 4.2.2 - Transform Step

The transform step allows flexible data cleaning and filtering. For instance, it permits modifications in the Record, Schema changes and creation or elimination of Records. Many other processes can be implemented at this point like, for instance, standardizing synonyms, acronyms and other equivalent forms of representation of data.

This step can also implement blocking by assigning Records to blocks. In the proposed framework, a Record is not restricted to belonging to a single block.

### 4.2.3 - Compare Step

This step deals with the comparisons between Records and the subsequent classification of pairs according to their similarity.  The user will define the two sources of Records, one of them being the reference source. Then all the pairs of records are looped through. For each pair, several matching algorithms are used to compare the fields in both Records, and a weighted score combining each field's score is computed. Two thresholds are defined by the user, the low threshold, bellow which pairs are classified as positively non-linked, and the high threshold, above which pairs are classified as positively linked. The remaining pairs, whose score is between the low and the high threshold, are classified as possibly linked. Then, the Results are created by evaluating the score of each pair. If blocking is used, this process is repeated for the records of every block, otherwise, all the records from each source are used in the comparison.

### 4.2.4 - Export Step

With the matching Results calculated, the export step deals with presenting such results in a convenient way. As an example, the export step can create a spreadsheet with statistics on the data set such as the percentage of duplicates, and the quality of the data, or a file with the unique identifiers of the duplicate Records, which can be used to merge the two sets of records in an external program.

### 4.2.5 - Matching Process

The matching process is a sequence of the previous steps, pictured in Figure 4-1. To begin, the data is extracted from its source and imported to the framework, in the form of Records. Then, the Records go through several transformations, so that Records from distinct sources are suitable to be compared against each other. A variable number of transformations can be chained together, allowing each individual transformation to be straightforward and have a single purpose. When the source transformations are completed, the Records are matched. This match produces the Results, which are finally exported to the user.

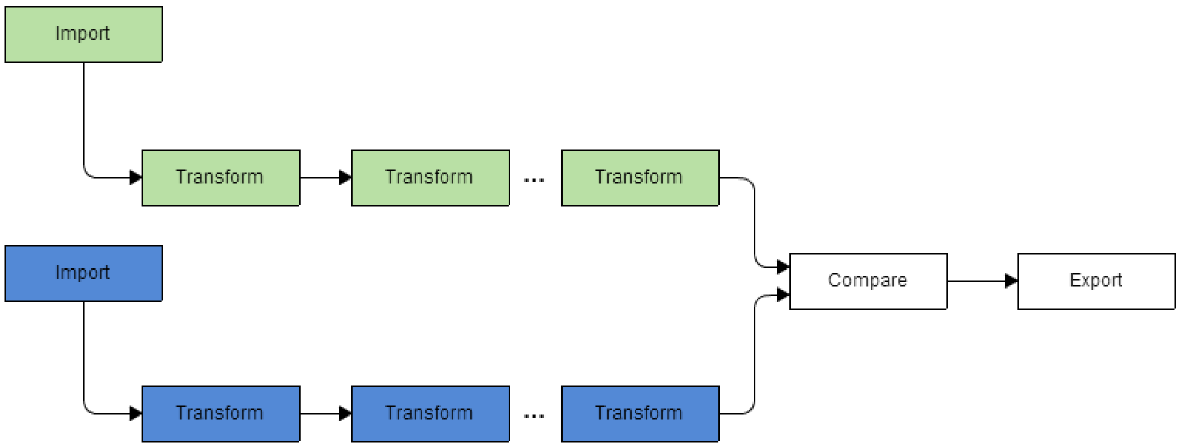

Figure 4-1 - Matching workflow. The colours differentiate between two data sources.

### 4.2.6 - Update Process

This process combines some of the steps previously described and deals with the provided data from an integration perspective. The purpose of this process is to add new records to a source that is already in the framework, assuring no duplicates are added.

The idea is to match a new record set against a reference data set (Figure 4-2). The user defines a pipeline of steps, determining the transform steps, compare step and export step.

The user then sends the new Records, which go through the transform steps. Then they are compared against the records that are already in the record set, designated as the reference records. The records that do not match any of the reference records are added to the reference data set. The subsequent Results are added to the Results previously calculated. These are then passed to the export step, which proceeds as previously explained.

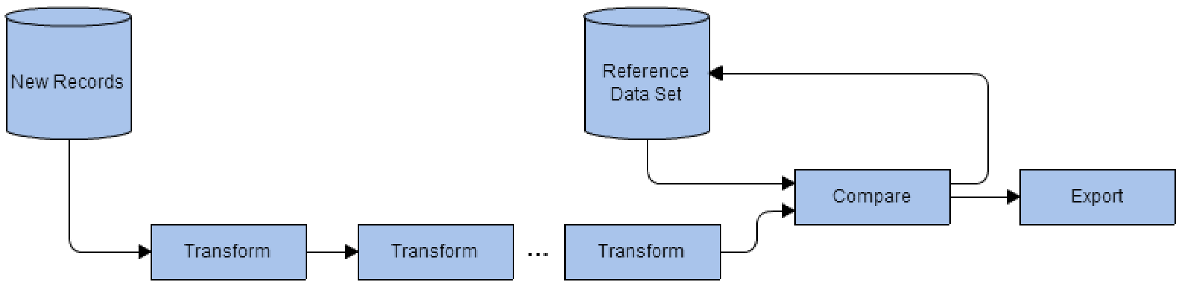

Figure 4-2 - Update process.

### 4.2.7 - Query step

The query step is designed to show the results of the matching process on demand. Rather than using only the Export step as a form of output, the user can also query the matches performed in previously executed matching tasks, accessing the matches produced by the Compare step, along with the matching records and scores. For this, the user provides a Record that serves as a reference for the matching process. However, this Record is not matched against Records in a dataset, but against the reference Records in the Result set produced after a Compare Step. This is useful to verify if a given Record has duplicate Records.

## *4.3 - Architecture*

We already discussed the various workflow steps necessary to provide a flexible record linkage solution. This section will describe how our framework implements such steps, while allowing efficient use of resources and generality. We will propose and discuss the architecture for a record linkage framework.

### 4.3.1 - User-Provided Plugins

As expressed, a plugin system allows an adjustment to the specificities of data. In the proposed architecture, the user can develop plugins that deal with the data directly, while the rest of the framework implements the abstract processes that are common between record linkage solutions. A plugin must implement a predefined interface to be valid in the framework context. There are several types of plugins associated to workflow steps, each of them with a different interface. However, they share two methods, one for initialization and the other for finalization. The first is called as soon as the plugin is needed, and should be used to prepare for the execution. This method receives a set of configuration strings, which the plugin is responsible for reading and adapting its behaviour accordingly. This is useful, once again, to allow a greater amount of generality in the framework. This way a plugin can have a set functionality but can adjust to different situations while requiring no modification to the code itself. The second method is called when the plugin concludes its work, to allow the release of any resources it might have used.

These plugins are to be packaged in a jar file so they can be used in the framework. Ahead, the purpose of each plugin is described.

**Import Plugin**

An import plugin is responsible for the reading tasks. It takes any kind of data source and transforms the proprietary data format into normalized Record objects, as described in the Import Step. For example, an import plugin might connect to a database and execute a query that selects all the records of a specific table. One of the configuration parameters would be the database location and access credentials, allowing the same plugin to be used for different databases. This behaviour could even be further generalized, if the configuration strings include the name of the table and the fields to select. In this example, the finalizing step would close the connection to the database.

After the plugin fetches the necessary data, it must then create the corresponding Record objects and pass them to the framework.

**Transform Plugin**

A transform plugin performs transformations on a set of records. These transformations can be of three types (Figure 4-3):

- Value-level – A value-level transformation modifies a specific value in the record. An example would be converting the name of a person into lowercase characters.
- Record-level – A record-level transformation modifies the structure (schema) of the record and the corresponding values. An example would be to split the full name of a person into the first, middle and last names, creating each of these fields and removing the field containing the name.

- Set-level – A set-level transformation uses several criteria to remove records or perform other operations on the set. An example would be removing the records referring to male individuals.

The transform operation outputs the processed set of Records, and the resulting Schema. Returning the updated Schema is necessary so that the next step can accurately manipulate the Records. The benefits of implementing a configuration method for plugins of this type are evident. For instance, a plugin to convert strings to lowercase can know which field should be converted and a plugin to filter records can obtain information about filtering fields and the filter expression.

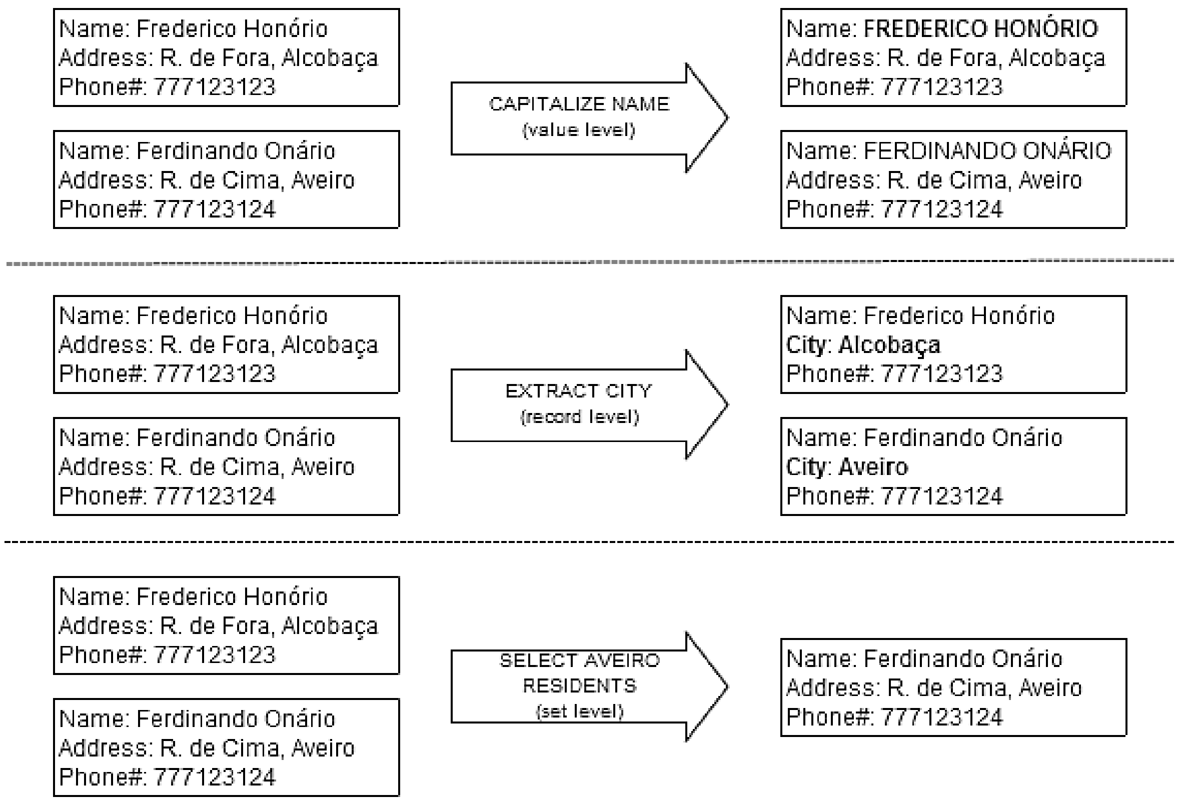

Figure 4-3 - Examples of transformations. The bold letters indicate the transformed values

### Compare Plugin

The compare step can use various algorithms in the process of determining a pair's score. These algorithms are implemented as compare plugins. These plugins simply receive two field values and compute a score of similitude, with scores ranging from zero to one. The mapping between a field and the compare plugin used is provided by the user as are the thresholds and field weights.

**Export Plugin**

The export plugin receives the Results created in the compare step and produces a result containing meaningful information to the user. The plugin could, for example, implement a XML file writer that would store the matches on a file, or execute a method in a web service for each match.

### 4.3.2 - Parallel record linkage

When analysing the workflow, we can see that the steps are discrete and self-contained because they only require the data from the previous step(s). In the background chapter the issue of the runtime complexity was raised. For matching without blocking the complexity is $O(N^2)$, and for transform steps, for instance, it is $O(N)$ because each Record has to be processed once. Although $O(N^2)$ is the worst-case scenario (because the runtime can be reduced by applying blocking), even $O(N)$ can be too prohibitive when dealing with large amounts of data. For reference, the average medium-sized healthcare can have 100,000 to 500,000 records, while a large facility can exceed 1,000,000 records [1]. For problems of this size, executing the workflow in a sequential order can result in runtimes that are too long to be useful. For this reason, exploring the possibility of executing record linkage steps in parallel is a justified endeavour.

Ahead we illustrate how some form of parallelization can be achieved for every record linkage step by dividing the work into smaller instances. We designate an executor as device that can perform these small instances of work.

- Import Step – If the access made by the user-provided import step is guaranteed to produce the same results for consecutive calls and the plugin can select a subset of data efficiently, then it is possible to run the import step as several smaller parallel steps. For this, the user needs to specify the total range of identifiers for the data, so that the framework can distribute this range through the available executors. The range of identifiers for an instance of the user-provided plugin is then passed by the configuration strings, which the plugin needs to read.
- Transform Step – Unless the user-provided transform plugin has set-level transformations, which might require the complete set of records to be properly evaluated, the execution of this type of plugin can be parallelized. If the user does not identify that the transformation requires the full set, the framework can divide the record set through the available executors.

- Compare Step – If blocking was applied by a transform step, then the framework can execute the comparisons for each block in parallel by distributing the blocks through the available executors.
- Export Step – As with the transform step, the user must also define if the export plugin requires the full set of Results or not, in which case the export process can be parallelized by dividing the Results by the executors.
- Update Step – Because the update step is designed to deal with a small number of new records, it is not necessary to partition this step.
- Query Step – Like the Update Step, the Query Step works with a small set of input Records and is therefore unnecessary to parallelize this step.

### 4.3.3 - Components

We'll now discuss the architecture developed to implement our solution. Having described a method to parallelize the execution of a record linkage process, it is clear that a distributed architecture is necessary, requiring the deployment of multiple executors. Furthermore, these executors need to be flexible to allow the use of the user provided plugins. With this in mind, we propose a system with three main components, the Master, the PluginSlot and the Repository (Figure 4-4).

The Master has four main functions. First, it acts as the main form of communication with the user. For this it implements a webservice that exposes the system's functionality. It also acts as a repository for the user-provided plugins which can be uploaded to the Master. Another important functionality is the division of the work of a record linkage step into discrete jobs that can be executed in parallel. Finally, the Master is connected to various PluginSlot instances, and allocates the parallel jobs to each instance based on the requests of the user.

The PluginSlot is the executor of the system. It is able to execute remote code that is dispatched by the Master.

Finally, the Repository maintains the data of the system. There can also be various instances of the Repository connected to the system.

The main form of communication between these components is the SOAP protocol. SOAP not only provides a means of communication but also encourages interoperability with other systems because it is a well-known standard. It is also important to promote a modular design, with well-defined interactions throughout the system. Finally, SOAP provides methods of authentication and secure transmission of messages. These were not implemented as they fall out of the scope, but are a possible point of improvement.

Another important issue is the way the user-provided plugins are used by the PluginSlot. By design, the user-provided plugins interact with data that is in the system but they never interact with any of the system components directly. Instead, they simply receive the data and return the result of the task. The exceptions to this are the Import and Export plugins, which interact with an external system for input and for output, respectively. However, communication with the system is never direct. It is then necessary to develop the routines that perform these generic operations. The simpler approach would be to implement such routines in the PluginSlot code, and choose which methods to execute based on the job given by the Master. However, our approach was different. Instead, the PluginSlot can execute a generic job (as opposed to being limited to record linkage jobs) and the record linkage routines are developed as Task Plugins. The Task Plugins then use the user-provided plugins to achieve the record linkage tasks.

The three components are further described in the following sections.

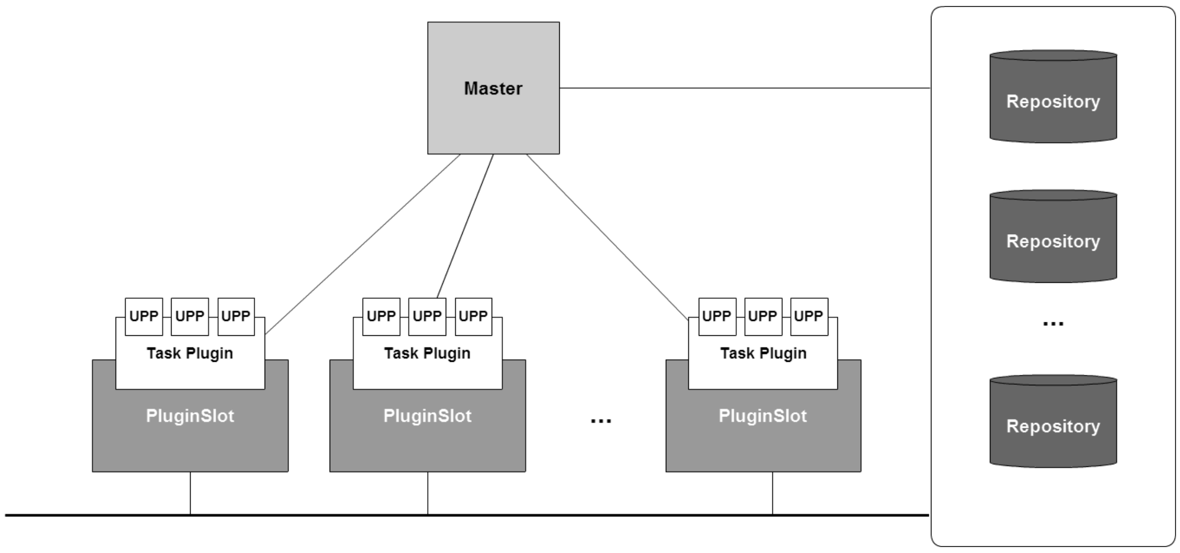

Figure 4-4 - System architecture. UPP designates user-provided plugins.

## 4.3.4 - PluginSlot

The PluginSlot can execute (part of) one step of the pipeline. Although it uses the user provided plugins, it does not execute them directly. Instead the PluginSlot is given a set of files containing a Task Plugin, and any other files necessary for execution. The Task Plugin represents a procedure that is finite. They are different from the user-provided plugins in that they implement a generic routine while the user-provided plugins are specific to record linkage and fit in the workflow. Listing 4-1 exemplifies the necessity of such plugins. It is apparent that very few instructions are dependent on the problem in question, and are necessary for the generality of record linkage problems. Only the operations performed by the compare plugins are problem-specific. This abstraction is also justified by the fact that without it user-plugins would need direct access to the database to perform their work, thus creating a possible security issue.

```
lev = PluginLoader.load("LevenshteinDistance")
dateComp = PluginLoader.load("DateCompare")
ref = Repository.loadSet("A")
oth = Repository.loadSet("B")
matches = [] //empty list of matches

for each record m in ref
      for each record n in oth
            scoreLev = lev.compare(m.name, n.name)
            scoreDate = dateComp.compare(m.date, n.date)
            score = scoreLev * 0.6 + scoreDate * 0.4 //weighted sum
            if (score > 0.8) //high threshold
                  matches.add(m, n, score)
repository.storeMatches(matches)
```

Listing 4-1 - Pseudocode of a simple compare method.

The Task Plugin interface has three methods (Listing 4-2). The first is a form of initialization, where plugin receives a set of configuration strings, much like the user-provided plugins. Along with the configuration strings, the plugin will also receive the execution context — contextual information about the state of the PluginSlot and the current execution. The second method is a processing routine that the plugin implements, which also receives configuration strings. This method can be called several times after the initialization and the configuration strings can be different for each call. Finally, a method to finalize the execution is also present, and should be used to free any resources taken.

```
      public    void    init(HashMap<String,    String>    settings,
HashMap<String, Object> context);
      public void process(HashMap<String, String> settings);
      public void destroy();
```

Listing 4-2 - Task Plugin interface.

To execute a plugin, the PluginSlot exposes a SOAP webservice that allows the execution of Tasks and the uploading of Task Plugins. A Task is the combination of a plugin, the method to execute (initialization, processing or finalization), and the settings for the method. The PluginSlot keeps a queue of Tasks, which are executed as soon as there is no running task, so an incoming task does not disrupt the running one. Once the PluginSlot has the necessary jar files to run the plugin, a class loader is used to load the plugin into memory, so its methods can be executed. Combining the class loading mechanism with the use of reflection, both provided by the Java SDK, allows the execution of remote code. Reflection is the ability to inspect an object at runtime, without previous knowledge of it. By using such capabilities, we can load an unknown class from a file into memory, inspect the methods that it implements and execute them.

The jar files might include other classes which also need to be loaded if they do not exist in the PluginSlot execution environment. Because of this, the classloader is passed as part of the execution context when the plugin is initialized, at which point it should load the desired classes and instantiate the corresponding objects. After the loading is complete, the three methods of the plugin can be executed remotely, by accessing the PluginSlot's webservice.

We have at this point a system to remotely execute programming tasks, now we discuss how this system can be used to solve record linkage problems.

**Record Linkage Task Plugins**

In order to use the PluginSlot for record linkage it is necessary to develop Task Plugins that implement the abstract behaviours identified in the workflow section, providing this way support to use the user-provided plugins and associated configurations. A brief description of the functionality of each Task Plugin will show how they fit in the framework:

- Import Task Plugin – Initializes the user-provided import plugin, executes it, and stores the produced Records in the Repository.
- Transform Task Plugin – Loads the Records that are to be transformed, loads the user-provided plugin, executes it, and stores the Records.
- Compare Task Plugin – Loads the Records from each source, compares every pair of Records by using the user-provided compare plugin and configuration, and stores the Results.

- Export Task Plugin – Loads the Results and executes the user-provided export plugin.
- Update Task Plugin – Loads both the old and new Records, transforms the new Records and matches the new Records against the old. Every successfully matched Record is added as a Match of the old reference Record while the Records that are not matched are added to the source, as reference Records.
- Query Task Plugin – Executes the transforms on the query Record, loads the Results, matches the Record with the reference Records of the Results. For every matching Result, the reference Record and matching Records are returned.

Because the Task Plugins always store the produced data in the Repository, the user is able to choose which set of Records or Results is used in the execution of a record linkage task.

With a better understanding of the Task Plugins' purpose, we can now discuss the necessity of having a generic executor, instead of a record linkage specific one. Mainly, the ability to run arbitrary code in the PluginSlot is useful because this component is responsible for varied and distinct tasks. It was previously explained that designing the PluginSlot to be completely task-agnostic was not crucial, the alternative being the integration of the abstract record linkage routines in the implementation of the PluginSlot. However, allowing non-specific tasks and later implementing the record linkage tasks offers more extensibility, enabling the possibility of adding other record linkage steps to the framework. In fact, the PluginSlot does not need to be confined to record linkage tasks because of its non-specific interface. The benefit for the maintainability and ease of development of the framework is also worthy of note. When errors in the implementation of the task plugins are found, they can be easily fixed by submitting a correct version of the plugin, and new functionalities can also be added easily. If the generic tasks were implemented directly in the PluginSlot, a complete redeployment of all the instances would be necessary.

### 4.3.5 - Repository

The Repository serves as a database, not only for persistence of the Results, but also as a "scratch area" where Records are kept between steps. It is used at the beginning and the end of most steps in the pipeline. The Repository uses MongoDB for persistence, a NoSQL database with some features that make it relevant for the framework:

- Flexible schema – Relational databases usually force the table schema to be defined beforehand. MongoDB (and other NoSQL solutions) can accommodate a flexible data structures without expecting a particular schema. Because the framework cannot anticipate the Schema that is used, it is practical that the chosen database allows a similar behaviour.
- Indexing – An index is used to improve the speed of data querying in databases, a necessary feature when dealing with large sets of data.
- Load balancing – MongoDB is able to split data efficiently through several instances, in a process called sharding, and distribute the requests through them to improve availability and response time.
- Replication – The database supports master-slave replication, allowing a master that provides the database functionality while the slave mirrors the data, serving as a backup.

As expected, the repository implements procedures to store and retrieve Records and Results. In addition, methods to present the number of Records or Results in a set are also implemented, so that the Master can accurately create parallel tasks. For this functionality, it exposes a SOAP webservice with several methods. A MongoDB client was implemented to connect to the database and perform operations on the form of queries, insertions and updates while also converting between the plain Java objects and MongoDB objects as needed.

**Querying**

Recalling the Query step, and given the large amount of data that can be involved, it is clear that matching a record against the complete set of records can be very time-consuming. Thus, a different approach to perform these queries is required. The query step is used to verify if a given record — specified by the user — has a match in the data set. Our approach relates to the blocking techniques already covered. When a Result is stored in the Repository, both the reference Record and the list matches — each match comprised of a matching Record and a score — are stored. With this, a Result indicates the relevant matches (above the high threshold) for a given record. The Result also includes the list of blocks the reference Record belonged to. This is the key point in our approach. Rather than performing the comparison on all the records, the provided record is transformed to be congruent with the remaining records in the framework. This transformation includes assigning the Record to one or several blocks. This assignment is only performed on the Record structure, in memory, and is not carried out to the framework. Then, the framework verifies which blocks the Record was placed in, and pulls the Results assigned to the same blocks from the database. The provided Record is then matched with the reference Records and matching Records of these Results. As expected, this method severely reduces the time necessary to provide matching information. Furthermore, as long as the blocking method is correct, i.e., if it does not separate similar Records into different blocks, this method can show the same accuracy as the exhaustive matching of the complete Record set.

### 4.3.6 - Master

The Master component controls the execution of the PluginSlot instances. It acts as a client to the PluginSlot instances by scheduling executions as necessary. The user is able to load user-provided plugins, configurations and other associated files. These parameters can be defined by accessing the Master webservice. It has knowledge of the existing PluginSlot instances and can use them to execute record linkage tasks. The webservice exposes methods to define the Task parameters, including the configuration strings for the plugins used and the name identifying the plugins. Defining the plugin and configuration triggers the creation of Tasks, which are accepted by the PluginSlots. The generation of Tasks depends on the availability of PluginSlots as the framework is able to parallelize the execution by dividing the work in multiple sub-Tasks according to the criteria described in section 4.3.2 - To be able to scale the work to the PluginSlots, the Master has to know the sizes of the Record sets and Result sets that are going to be processed, requiring then a connection to the Repository.

Given a way to distribute the work by various sub-Tasks, the problem of executing the Tasks in the correct order arises. This issue can be solved with a directed acyclic graph structure (Figure 4-5) to represent the dependencies between tasks, where the nodes represent the tasks. The graph is directed so that the direction of the edge can indicate the relation of dependence, that is, an edge from $A$ to $B$ denotes that $B$ depends on $A$. The acyclic property guarantees that no node can follow a sequence of edges that lead to itself eventually. This is useful to prevent deadlocks, otherwise, if $A$ depends on $B$, $B$ on C, and $C$ on $A$, there is no task that can be ever executed because every task is waiting on another one to finish. When the user requests the execution of a step, the Tasks are added to the graph. If the Task depends on a different Task already in the graph, it is added as a child node. This way, the nodes of the first level contain the Tasks ready to be forwarded to the

PluginSlots. Even though the Tasks can be divided into multiple sub-Tasks, they are not all placed in the graph. Because the execution of the next Task requires the conclusion of the previous sub-Tasks, it is sufficient to represent the overall Task with a sub-Task counter. The counter represents the number of sub-Tasks that are not yet finished and is decreased when a PluginSlot finishes the execution of a sub-Task. When all sub-Tasks are finished, and the counter reaches zero, the Task is removed from the graph and the next Task is connected to the root of the graph, indicating that it is ready to be executed.

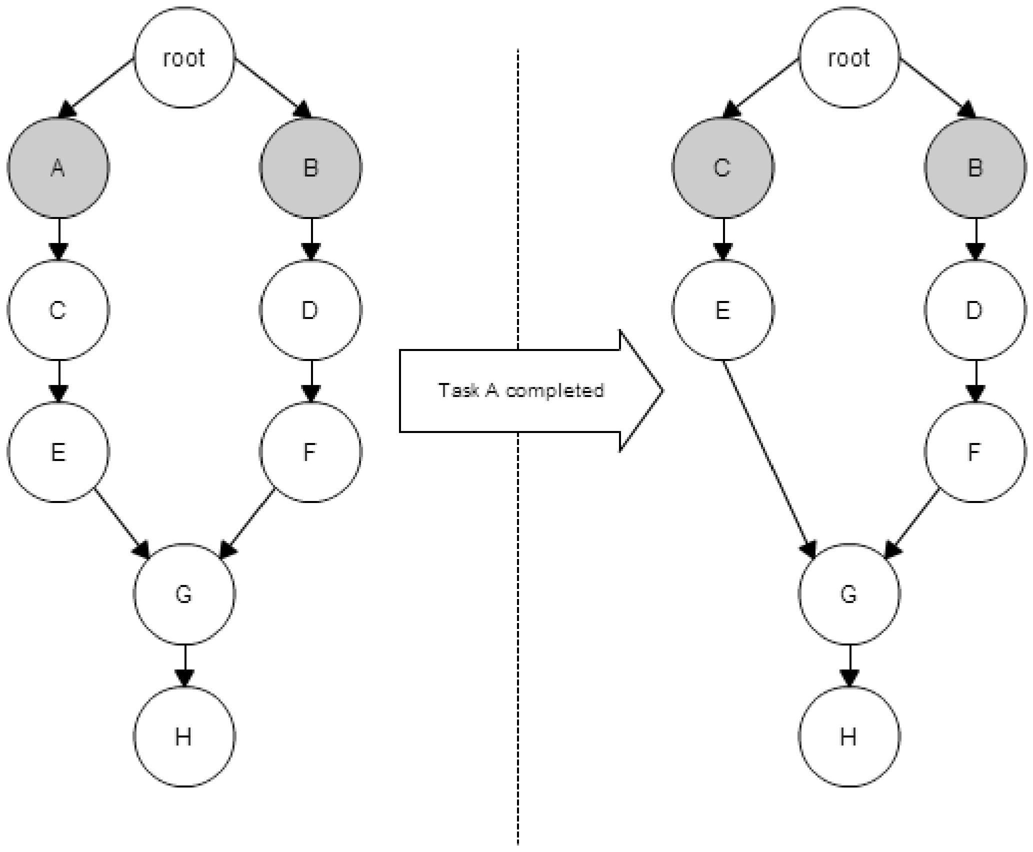

Figure 4-5 - Representation of task dependencies, showing the process of removing a task. The grey nodes identify tasks that ready to be executed.

Another issue is the way in which the Tasks are allocated to the PluginSlots. There are several methods of scheduling jobs in distributed system. However, scheduling methods that require an estimate on the execution time of a Task are not feasible in this scenario. This is because the execution time of a Task is heavily dependent on the code executed in the plugin. So, it was decided to use a simple scheduling solution. However, the framework has flexibility to support more complex algorithms. In our approach, with *N* PluginSlots connected to the Master, the work is divided into *N* sub-Tasks and a single Task is forwarded to every idle PluginSlot. The remaining Tasks remain in a queue and are forwarded once PluginSlots become available.

Finally, part of the Task definition is its domain. This is a sequence of characters that identifies the problem that a group of Tasks is solving. It should also identify the data sources, namely the sets of Records. With this identifier, the framework can know which sets of Records are accessible to a Task, and restrict access. However, this setting must not be the single method of authentication. While it is able to restrict the execution of Tasks to a related group of Records, it does not prevent a third-party to use the same domain string. Therefore, the user must generate an identifier string, which is unique within the system and restrict its access (knowledge) to users working on the same problem. Because the framework was designed to be used by other systems, creating a system of access control and user authentication was not pertinent to this work, and this simpler solution was chosen instead.

To submit the Tasks to the PluginSlots, the Master implements a client to their webservice. Before submitting the Task, however, the Task Plugins, user-provided plugins and other files are sent, and only then the command to initiate the Task is issued.

## *4.4 - Implementation*

In this section, some of the choices made during the development are described in detail. They are described because they highlight important aspects of the design that support the stated requirements.

### 4.4.1 - Record

The design of the Record is of great importance as it is extensively used throughout the framework. It is also important that the design is clear and efficient, because it has to be used by the user-provided plugin developers.

The Record is the structure that contains information for a single entity. It consists of a list of values and a list of blocking associations. The list of values simply contains the values for each field that is described in the Schema, in the same order, much like a row in a spreadsheet. The blocking associations are represented by a list of block names, and every Record is considered to be in a block if it has its name in the list.

The need to separate the Schema from the Record was mentioned previously. An immediate solution is the use of a hash map structure. This acts as a dictionary, allowing a value to be paired with a key, which is used to access the value. The problem with this solution is the overhead in memory used in runtime, as it requires a greedy allocation of memory that vastly surpasses the amount of memory the field values would take if stored sequentially in an array. However, being able to refer to values by their name is a useful feature from a programming standpoint, and by using an array the user will not have this feature and is required to know which index corresponds to a given field.

The initial design had a reference to the Schema within the Record and the values kept in an array, a solution which would allow a straightforward use of the Record. For example, the Record could have a method to set a value:

```
void set(String fieldName, String value);
```

because it could access the Schema and verify which is the index the field name is stored in. The issue with this approach is transferring the Records. While programming an import plugin for example, the user can create a Schema object and use the same object for every Record, meaning only a single instance of the Schema exists in memory, which is desired. However, when transferring the Record through SOAP webservices, the marshalling process would duplicate the Schema every time, even if it is consistent across Records. For this reason, the Record is represented as list of values, without a Schema directly associated. There still remains the problem of using such a structure effectively, the reason for which the RecordHelper was created.

The RecordHelper is a utility class to make the creation and manipulation of Record objects more accessible to the developer. The RecordHelper requires the Schema of the manipulated Records to be provided, so that it can identify the fields of the Record. These are some relevant methods:

```
public Record newRecord();
public String get(Record record, String field);
public void set(Record record, String field, String value);
```

Listing 4-3 - RecordHelper methods.

The `newRecord` method shows the creation of a Record. Although creating an object is usually simple, in this case the user has to assure that the Records are consistent within a step. Because the Records are represented as a list of values, it is possible to have Records in a set that don’t match structurally, if the user allocates the lists manually. The use of the `newRecord` assures this does not happen. Furthermore, the `get` method returns the value of the indicated field, while the `set` method defines the value of a field. The combination of the Record, Schema and RecordHelper structures compose a solution that is efficient in terms of memory and network transmission and maintains a degree of usability for the developers that interact with such structures.

Another important concern is the method that is used to represent blocking in the framework. Blocking is achieved by appending the name of the block to the list of block names. If the Record is contained in a list (or a Record set), the list itself is not modified. A different option would be to assign the Record to a list, and have a list for each of the blocks. This would make programming a plugin that performs blocking more difficult because the plugin would have to maintain various lists, and return these lists at completion. The chosen approach is simpler to developers: when the plugin determines the Record belongs in a block, it simply calls a method in the Record to assign such block. The method in turn adds the block to the list of blocks. However, this approach is only viable because of the method of storage . When stored in the Repository, the list of blocks is kept as special field that is distinguished from the other fields of the Record. We can use this method because MongoDB allows indexing based on fields, which reduces the time necessary to query the database on an indexed field. This way, we can define the blocking field as an indexed field, and have the results in a short amount of time.

Finally, is also relevant to discuss an issue encountered during development. To develop a client for a SOAP webservice in Java, developers often use the wsimport tool, or an application that uses it internally. This tool is useful because it creates several classes (artifacts) that serve as client to a SOAP webservice. These transactions often involve objects, so it is necessary to invoke a webservice method with objects as arguments. This is a problem because the developer would require access to the classes that are received as arguments, to properly construct objects that the webservice can manipulate. For instance, suppose a webservice provides a method like so:

```
int getAge(Person p)
```

This method returns the age of a person, in years, by reading the data of birth. A client to this webservice would have to know how to create the class Person and how to define the date of birth. Wsimport solves this issue by creating class stubs. Stubs are classes that mimic the data structure of the original class, but don’t carry the implementation. This means the developer can correctly fill the stub object with data, and communicate with the webservice with said object. While this is a useful feature because it requires minimal knowledge of the internal classes of the webservice, it creates an issue for our framework. In this case, the Records are passed through various webservices, but all of them are internal, and the generation of stubs is not necessary. In fact, because the stub is technically a different class, it can not be used interchangeably with the webservice unless some modifications are made. This problem was met when developing the abstract record linkage routines that integrate the user-provided plugins in the framework. For a transform Task Plugin, the Records have to be fetched and stored in the Repository. If the Repository client is generated with wsimport, the Records involved in the transactions have to be stubs. However, the user-provided transform plugin is expecting a list of the original Records and the method can not proceed. Through some experimentation, it was discovered that removing the generated stubs was sufficient to force the webservice client to use the original classes. By configuring Ant, the build system used during development of the framework, we were able to remove these files seamlessly during the build process.

### 4.4.2 - Sandboxing

Having the ability to execute code provided by users is a core feature of the system, however, it poses some security risks.

One of the security risks is the ability to interfere with Records of other users or other steps. This is combated by the design of the framework. Each user works within a domain, and access to other domains is impeded by the framework. This means users can not access Records or Results from other domains. Within a domain, a user can access the Records through the various plugins, however, this is not direct access, simply the ability to read the data, not modify it. Because the various Records and Results are persisted separately between each step, no plugin can affect with the data from other steps, except from reading it. Every file that is received is placed in a directory named by the domain of the task. Once the PluginSlot receives a new task, the previous domain directory is deleted, and thus every file used in the previous task is removed.

The other risk is interfering with the hardware itself. Again, the design of the framework provides some form of control. A user with malicious intent can create a plugin to continuously allocate memory or work in an endless loop. This risk is unavoidable when dealing with compiled programs coming from an outside source. However, these malicious tasks will only affect the PluginSlot instance the plugin is running on, and will have no effect on the rest of the system because the PluginSlot is the only component which executes plugins.

## 4.5 - Plugin Development and usage

Integrating user developed plugins in the framework is a simple process. To develop plugins, users require the RL-Common package. This package contains the necessary data structures and interfaces that are used within the framework which can be manipulated by plugins. For a plugin to be used in the framework, it has to implement one of the provided interfaces. Plugins must also implement the initialization and finalization methods:

```
void init(HashMap<String, String> settings);
void destroy();
```

We see that the `init` method can use a HashMap to receive configuration strings. This data is set by the user, when requesting the execution of a transform step. Therefore, there must be an agreement between the user and the developer of the plugins, because the names of the fields must match, and the plugin must interpret them accordingly.
The specific methods for each plugin are now described.

For the Import Plugin:

```
public RecordSet import();
```

The plugin must create a Schema object defining which fields will be stored in the Records, and also create a RecordSet. A RecordSet is a composed of a list of Records and the associated Schema. The Schema should be ascribed to the RecordSet and the Records added to it as they are extracted from the source. The RecordSet also has a name, which the plugin must set and is used for referencing in various steps.

The Transform Plugin has the following interface:

```
public List<Record> transform(List<Record> records, Schema schema, String[] fields);
```

Instead of creating a RecordSet, the transform plugin should return a list of Records, this is because the framework deals with the results of transform steps as versions of the original RecordSet. The provided Schema refers to the list of Records, and should be modified in case the plugin creates Records with a different structure. The array of fields identifies which fields should be modified in the operation. However, the plugin is not restricted by this information, being able to modify whichever data it chooses to. It enables the plugin to be more generic, being able to perform an operation on fields without knowledge of their name while being developed.

The Compare Plugin has this interface:

```
public double compare(String valueA, String valueB);
```

The compare plugin should compare the two values and produce a score between 0 and 1. There is no guarantee on the order of the comparison, so the operation should be commutative, that is, comparing $A$ with $B$ should produce the same score as comparing $B$ with $A$.

The Export Plugin has the following method:

```
public void export(List<Result> results);
```

The export plugin simply receives the Results of the matching, and can perform an external action, for every result.

The final step in creating a plugin is packaging and deployment in the framework. The developer must then compile a jar file containing the plugin. Then, using a webservice method provided by the Master, the user can upload the plugin and identify it with a name.

Using the plugin in a step is then a matter of calling the appropriate webservice method (there is one for every step) and providing any configuration necessary. This includes the user-provided plugins used in the step, the Record sets involved and other parameters. Along with the configuration of a step (e.g. the Record set for a transformation or the thresholds for the comparison), the configuration of the user-provided plugins may also be provided.

# Chapter 5 - Discussion and Results

This chapter aims to discuss the merits and results of the proposed framework. Having defined the requirements for a generic record linkage tool, we can now validate this work, and highlight its contributions.

The ability to interact with external data sources is enabled by the user-provided plugins. Developers can implement routines to extract data from a wide variety of data sources, as long as they are accessible by the framework. The same generality is also present for the export of data. By developing export plugins, developers can present the results of the linkage process in various ways.

Allowing for dynamic sources of data is also an important feature. Matching static sets of records may not be a practical scenario in systems that have to be consistently active because the data in these systems is constantly changing. The update step allows users to define a pipeline of plugins to perform record linkage. This pipeline can then be executed as records are sent to the framework. This feature allows integration of the framework in a more complex system that has record linkage requirements which is important in this context. In the medical field for example, record linkage is one of the many tools to maintain data quality. Because of this, allowing live updates on the record set is crucial to expand the usability of the framework, as it allows efficient integration with other systems.

The extensibility provided by the plugin system greatly enhances the framework. The system can easily adapt to new techniques of record linkage, requiring only the development of a user-provided plugin for such techniques. Because the interaction with external systems can also be made by plugins, the framework is able to adapt to new methods of storage. Moreover, this extensibility goes beyond the integration of user-developed plugins and is also applied to the framework architecture. This means that the addition of different steps (not specifically record linkage steps) is also simple.

By using the proposed framework architecture, the user is able to benchmark various techniques by implementing different plugins. For example, it is possible to test which edit distance metric produces the best results, without having to repeat the previous steps (import and transform). This is feasible because the data that is produced by each step is maintained in the Repository and can be reused. For instance, it is possible to use various phonetic algorithms in the transform step, and verify which fare better when compared with. The expressed framework flexibility is of great importance when developing a solution for a record linkage problem.

The ability to execute Tasks in parallel, coupled with the fact that multiple PluginSlots can be used simultaneously, indicates that the tool can scale horizontally by adding more PluginSlots. However, the user is required to design a solution where the Tasks satisfy the stated parallelization requirements. Having illustrated how much data record linkage problems involve, we believe that this is an important part of this work.

The modular design of proposed framework also means that the components can be replaced or improved with little effort. Moreover, new steps in the pipeline could also be added easily, as the PluginSlot does not need to anticipate the type of task received.

The requirement of security and confidentiality is also satisfied by the design. Given that only the Task Plugins can directly access to the Repositories data, the protection of the data in the framework is assured. There is no assurance that a plugin will not produce incorrect data, but it can never tamper with the data that is produced by other plugins. Privacy is also assured by a domain identification mechanism that limits unauthorized access to data.

We also believe that the framework is very suitable for deployment in a SaaS context. In this model, the system is hosted on the cloud, and the user is provided with a service instead of a software package. With the proposed architecture, it is easy to envision a multiple concurrent users performing record linkage tasks in an online service. SaaS systems are usually distributed systems that are able to scale on demand by adding more hardware to provide processing power. Our PluginSlots system easily fits this description, supporting an SaaS-able architecture. The SaaS model also provides convenience for the user and is a good model for this framework. As illustrated, the framework is able to handle larger problems by performing record linkage tasks in parallel. This ability comes at a cost, however, as the parallelization implies an increasing cost in hardware to deploy multiple PluginSlots. Unless the user requires the continuous execution of these tasks, this cost may not be justified. However, in a SaaS model, this cost is covered by a single third-party, that can provide these services to multiple users and have a more efficient use of the hardware. For the framework, a SaaS model would also allow collaboration in regards to plugin development. Because there are many users of the system, the developed plugins can be shared. The result is a system with a wide range of out-of-the-box functionality.

## *5.1 - Assessment use cases*

We can now explore the functionality of the system and study its applicability to record linkage problems. The approach to record linkage problems is scarcely trivial as the process often requires testing, clerical review of the results and multiple iterations of development. In addition, these problems often involve data or services in other complex systems. To demonstrate the flexibility of the framework, we now present some possibilities of integration with different systems.

### 5.1.1 - Usage in ETL processes

A possible case of usage of the platform is to facilitate ETL processes. For this, the platform can be used in conjunction with Kettle. In this example the user needs to merge product data from two distinct warehouses, in order to use a centralized database but maintain the data previously acquired. Warehouse A uses a spreadsheet to maintain the inventory. Warehouse B maintains stock information with the help of inventory management software that can export a XML file with said information. Both sources have a local identification number, product name, vendor and description, along with other fields. The local identification number is a unique identifier of a product, but is only relevant within the product warehouse. That means the same product can have different identification depending on the warehouse it is catalogued in. The names of the product and vendor are consistent in both sources, but the spreadsheet method is more susceptible to input errors, ruling out the option of using exact matching.

To begin, the user must implement the import plugins to read the spreadsheet and the XML file.

For the first case the user can resort to a Java library to read spreadsheet files. The resulting plugin accepts a string with the name of the file to read. The initialization method simply reads this string and stores it to be used later.

The processing method creates the RecordSet and the corresponding Schema. Then, the method reads the file, iterates through every line and reads its values, placing them in a new Record. The framework will not be merging the Records directly –that task will be performed by Kettle – so only the values that are relevant to the matching process are required. Every Record is subsequently inserted into the RecordSet. Finally the file is closed. This enables the usage of XML file data in the framework.

The plugin can then be uploaded to the framework by using the webservice. The user needs to compile the plugin into a jar file. Then, the user can call the upload method and provide the binary content of the jar file and a plugin name as argument, making the plugin available to the framework.

A similar process would occur for the XML file, where the contents of the file would be read to produce the Record set of Warehouse B.

The user would then instruct the framework to create two import tasks, for each of the sources and choose the plugin to run the import. We can say that user may simplify the process by creating the Records for each database with the same Schema, forgoing the need to implement any transform steps. By using the configuration settings, the user would define that only the product name, vendor and local identification would be extracted.

To perform the compare step, the user could simply use and edit distance metric to compare each field. The user would then determine the adequate weights and thresholds. A compare task with this information would then be submitted to the framework.

Finally the user would need to develop an Export Plugin to create a file with the mapping between warehouse identification numbers, which would be provided to Kettle. Then, a Kettle transformation would have to be created. This transformation would be able to read the data from both sources, and in addition read the file with the mappings created by the framework. Then, it would read this file and begin to stream the data from the database files. Whenever record is found whose identifier is present in the mappings, this record is placed into a special queue, waiting for the matching file from the other database. Once the pair of files is in said queue, it can be sent to a diverging path in the transformation, where the merging is done. The records that are not present in the mappings go to the intended transformation until they are inserted into the new database that contains all the records. Once the merging of the matched records is complete, these also go the transformation that eventually adds them to the new database.

### 5.1.2 - Neji

The flexibility of the PluginSlot has been previously mentioned. Because the Task Plugins have an abstract interface, the associated process can be very broad — as demonstrated by the ability to execute different steps of the record linkage process. However, the applications of the system can go beyond record linkage.

To demonstrate this flexibility, we describe the integration with a different framework. Neji is a framework for concept recognition for biomedical purposes, developed by the bioinformatics group at the Aveiro University. An article describing Neji is still being revised. The task of concept recognition is concerned with the identification of biomedical concepts in texts, and the categorization of such concepts. For this, it not only has to apply methods of NLP (natural language processing) — to separate the individual elements of a sentence — but also recognition in the form of dictionary matching and machine learning-based solutions.

The framework is composed of several modules (which can be complemented by the users), and users are able to define a pipeline of modules to execute. The framework is capable of scaling vertically, i.e., utilizing parallel processing by using multiple processors or processor cores to improve the execution time. However, methods of distributed execution, using networked computers, are not implemented. Neji is also able to deal with large amounts of texts. Once defined a pipeline of modules, the user can point to an input directory where Neji reads the included files. It is at this point that several execution threads are created to process the files.

Although this task is not specifically record linkage-related, we are able to use proposed framework to provide a distributed processing of Neji tasks. The pipelined nature of Neji is relevant for this type of application. Because the processing of a file through Neji is an isolated task, we can distribute the files in the same manner as we distributed Records in a transform step. A small modification of the Master was necessary, to allow the definition of the Neji pipeline through the webservice. Additionally, the distribution of Tasks cannot be based on identifiers, and the filenames are used instead. Thus, the user uploads the files to process to the Master, and defines the pipeline, enumerating also the names of the files, which is used by the Master to divide the work. The list of files is divided through the PluginSlots, and each file is uploaded to the attributed PluginSlots, along with the Neji Task Plugin. The Neji Task Plugin uses the initialization function to read the definition of the pipeline. Then, the processing method simply runs Neji, indicating the domain directory as the input directory. This way, we are able to distribute the processing of multiple files through several parallel executions. The Neji Task Plugin jar file includes a class implementing the Task Plugin interface, the Neji framework and any other module that might be used for this particular Neji pipeline.

The effort required to perform this integration was small in relation to the benefits gained. Neji is now able to scale horizontally. This means that Neji can be made to process large numbers of files efficiently, by adding more instances of PluginSlots to the system. We believe this application of the proposed framework highlights the extensibility and modularity of the design, as well as the capacity to scale to large problems.

### 5.1.3 - PatientMapper

Another example of usage is the integration of the framework into a larger system. This use case builds an EMPI (Enterprise Master Patient Index), a database with information pertaining to medical patients. It is focused on interoperability between systems of distinct healthcare departments. An article describing this system in depth is being reviewed. To maintain such interoperability, several standardized workflows were implemented, namely the PIX (Patient Index Cross-Referencing) and XDS-I (Cross-enterprise Document Sharing for Imaging) procedures. PIX defines a standard for cross-referencing patient identifiers maintained by different entities. XDS-I deals with sharing imaging documents (MRI, CT scan, etc.) between parties. The system is able to receive imaging documents from the imaging department of a medical institution. Along with the document, information pertaining to the patient is sent, including the patient identifier in the imaging department. The EMPI task is to link the patient identifier assigned by the imaging department to a global identifier. It is at this point that our tool is used. As the purpose is to relate patient identifiers, it is referred to as the PatientMapper within the EMPI system. The patient information is extracted from the document, transformed into a Record and verified against the existing Records in the database through a query step, using Levenshtein distance on the fields. Depending on the result of this match, three classifications can be given to the Record: new patient (positive non-link), linked patient (positive link), weak-linked patient (possible link). If a positive link is found, a message to merge the duplicated patient information is propagated. The possible match requires the intervention of a physician, to confirm the person referred by the two Records is the same. The ease of integration with our framework is aided by the usage of SOAP protocol, a mainstay in the medical information field.

With this application, we demonstrate the capability of integration with other platforms and the contribution of a generic record linkage platform to a field of much importance. The code of the Levenshtein plugin is presented in Appendix B.

## *5.2 - Results*

This section exposes some quantitative results of our work, specifically where performance is concerned. It is important to mention that the important contribution of this work is the framework concept and architecture. This means that although these results are relevant to indicate the outcome of our work, they do not define an upper limit to the performance, and can certainly be improved.

### 5.2.1 - Task performance and overhead

We have discussed the advantages the framework presents for performing record linkage. However, the overhead in processing time was not analysed. We can assume that a solution designed for a specific problem will have better performance because it does not implement our level of abstraction. We now attempt to measure these differences.

For this, we focused on the more complex step, the update step. This step can potentially have a longer running time because it involves transformations, matching of records and the output of results. This is countered by the small number of records the step should take at input. However, there is no restriction applied by the framework. To benchmark the steps execution in the workflow, we modified the Update Task Plugin to output the system time at specific points in the execution. Our case study uses a randomly generated set of medical records, which include a global (and accurate) identifier, the name, gender and date of birth with 10,000 records in total. We also included duplicated entities with variations in the values to ensure matches are found. The export step was omitted in this instance. The defined pipeline includes a transform step that converts the values in the record to uppercase and the matching configuration. Matching is done by using the Levenshtein distance on the first, middle and last names and exact matching on the gender. Field weights are the following:

- First name: 0.33(3)
- Middle name(s): 0.16(6)
- Last name: 0.33(3)
- Gender: 0.16(6)

The low and high thresholds are 0.50 and 0.85 respectively. These configuration values were attained by having multiple runs and maximizing the number of correct matches. In this scenario, we used only one instance of the PluginSlot. The Master and the PluginSlot were deployed on separate machines with Intel Core i3 2.3 GHz and 4 GB of RAM. These runs were executed ten times. The execution times of several sections of the process are as follows:

| *Section* | *Average time* | *Standard deviation* |
|---|---|---|
| Task submission | 4.6 seconds | 1.1 seconds |
| Record set transformation | 2.2 seconds | 0.5 seconds |
| Record matching | 2 minutes, 27.2 seconds | 1.5 seconds |

Table 5-1 - Execution times of an Update step.

We see that although the task submission time is high when compared to the transformation, it is almost negligible when dealing with matching, which is the core issue. These results show that even with a focus on generic record linkage, the cost in providing the required abstraction is not impactful on the performance.

With this configuration 455 of the 477 matches were found, resulting in 95.4% accuracy. The possible matches were 413,376, which included the remaining 22 matches that were not found. The remaining 4,999,536,169 pairs were classified as non-matches. While the matching results are not relevant because they depend on the matching configuration and plugins, they are useful to illustrate the ease of use of the tool. It is relevant to indicate that any measure concerning the matching capabilities of the framework varies heavily with the choice of user-provided plugins and configurations. Some of the plugins developed can be found in Appendix B.

Finally, these results allow for an estimation of the complete process runtime. We were able to match 10,000 in less than 3 minutes. Consider then a database with 500,000 records with similar information. In similar circumstances and with the same configuration, the process would be completed in two hours and thirty minutes with a single instance and be reduced to 6 minutes if 25 instances were deployed. These values show the importance of having a tool that can scale horizontally to use new hardware, allowing the process to be completed much sooner.

Also consider that the hardware used to make these measurements is in no way high-end, and more powerful materials would decrease the time to process a single block, and therefore hasten the whole process.

### 5.2.2 - Querying performance

Having described the execution performance, it is now important to verify the ability to query the Repository. The used hardware is similar to the previous test.

For this test, a similar generated record set was used, comprising of 500,000 records. The blocking approach relied on selecting the first three letters of the full name, and uses them for blocking. This creates the blocks AAA, AAB, and so forth. The names are capitalized first in a transform step. Finally, the records are matched. The matching is based on the measurement of the Levenshtein distance, and only the full name is used (its weight is 1). The low and high thresholds are 0.5 and 0.85 respectively. Once the matching is complete, we access the webservice and provide a record as the query. This record is processed and attributed to a block, based on the first letters of the name, and this block is pulled from the database. We measure the time necessary to extract the complete block and the time to match the block's records with the provided record. We verified which block had the largest number of records, and queried records in that block.

The largest block had around 22.500 records and was fetched in less than one second, on average. The matching of the queried record with the remaining records in the block took 300 milliseconds, on average.

This means that with data of this size, it is possible to have a response time that is short enough to be used by other systems in a live decision making process, further reinforcing the flexibility of the framework.

# Chapter 6 - Conclusions and Future Work

In this chapter we summarize the work developed. We also discuss the main contributions of this work, as well as opportunities for future research.

We began by researching the state-of-the-art in record linkage. The common techniques and methods were investigated, including block, data cleaning and probabilistic matching. The probabilistic approach was chosen to be the foundation of this work.

After this, several software solutions, both commercial and otherwise, were investigated. These provided insight to the problems involving record linkage as well as examples of usage.

Then, after surveying the available tools and the processes involved in record linkage, we set out to define the requirements of our tool. Thus, we aimed to create a framework able to perform generic record linkage, capable to interact with other systems, prepared for dynamic additions to a data set, configurable, extendable and, at the same time, secure, robust and scalable.

Enabled by our study of the state-of-the-art, we were able to propose a workflow for generic record linkage. That is, we described several steps that, when combined, can describe a record linkage solution that is compatible with most record linkage problems.

This was followed by the design of a software framework were the workflow would be performed, maintaining the defined requirements. We aimed to keep the design consistent with our ideal of generality and lessen the impact of large data sets. Also essential was the ability to utilize plugins in the execution of the record linkage workflow. For this, we designed a simple executor, the PluginSlot, which is able to perform general programing tasks. Then, we devised several methods of dividing the work of a workflow step into smaller units. We then developed Task Plugins that implement these steps in a parallel manner. The Master is the central system component that manages the deployed PluginSlots and interacts with the user. The Task Plugins are forwarded to the PluginSlot, and are able to use the user-provided plugins for record linkage. The user-provided plugins are responsible for manipulating data specifically. Such a design means that the framework requires no knowledge of the characteristics of the data, allowing the user-provided plugins to make direct manipulations. This design also means that, because the PluginSlot can accept any Task Plugin, new steps in the workflow can be introduced. In combination with the Master, several active PluginSlots are able to compute record linkage steps in parallel, reducing the execution time. A database, named Repository, was also developed to store the data in the framework. We also explained some issues encountered during development, and the solutions put in practice.

Finally, the results show various forms of integration with and within other platforms, emphasizing the interoperability. Moreover, several performance measures were made. First, we show that even though some overhead is observed, it is not significant when compared with the time of record linkage tasks. Also, some speculations are made about the processing time of large datasets. From these we reinforce the importance of our distribution mechanism, which allows tasks to be executed concurrently. Our storage mechanism, the Repository, was also tested for performance, and we concluded that the querying provided by our system could be used in a system with near real-time requisites.

## *6.1 - Contributions*

The highlighted contribution of this work is the framework design. We have discussed how record linkage is a tool used in many areas, and the flexibility of the proposed design is significant. Furthermore, the ease of development and analysis with our tool are crucial. With our benchmarking approach, analysts are able to revise their methods efficiently, whether by improving on matching algorithms or by tuning the parameters of the probabilistic matching. We also show the importance of confidentiality and described our methods to achieve this.

The techniques of parallelization described are also worthy of note. We describe several methods to divide a record linkage task into smaller task that can be executed in parallel. This fact is important because, as we mentioned, record linkage tasks are often applied to large data sets. With this parallelization, the framework is able to complete the record linkage process in a smaller amount of time.

## *6.2 - Future work*

Despite our results, much more exploration can be made. We now expose some avenues for research and improvement to the framework:

- *Implementing a user interface* — We described the framework as interoperable with other systems. However, the framework lacks a user interface. Designing a graphical interface to expose the described functionality would improve user experience and usability. It could also involve a system of authentication.
- *Improving the Master* — Although our architecture is distributed, the Master is a single point of failure. That means that the overall performance of the framework is heavily reliant on this component. Furthermore, a test to validate the ability of this component to manage large numbers of PluginSlots was not performed, as it would be too costly. Methods to allow multiple redundant Masters could be investigated.
- *Private record linkage* — Development of private record linkage techniques is also significant. The framework would allow the addition of these techniques, either by the addition of a private record linkage step which would perform the matching of private records, or the use of a private matching function that fits in our probabilistic model by comparing private fields and producing a score.
- *Sandboxing and secure execution* — Another issue raised during the examination of the PluginSlot is the secure execution. As it stands, a malicious plugin can easily halt execution by entering an infinite loop or allocating excessive amounts of memory. Methods to inspect the activities of a plugin or otherwise restricting the execution would be a good addition, especially if the framework was to be used in a SaaS environment.

# Chapter 7 - References

[1] K. Williams, K. Robinson, and A. Toth, "Data quality maintenance of the Patient Master Index (PMI): a 'snap-shot' of public healthcare facility PMI data quality and linkage activities.," *The HIM journal*, vol. 35, no. 1, pp. 10–26, Jan. 2006.

[2] P. Christen, "A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 9, pp. 1537–1555, Sep. 2012.

[3] O. Charif, H. Omrani, O. Klein, M. Schneider, and P. Trigano, "A method and a tool for geocoding and record linkage," *2010 Second IITA International Conference on Geoscience and Remote Sensing*, pp. 356–359, Aug. 2010.

[4] S. a Kinner, S. Forsyth, and G. Williams, "Systematic review of record linkage studies of mortality in ex-prisoners: why (good) methods matter.," *Addiction (Abingdon, England)*, vol. 108, no. 1, pp. 38–49, Jan. 2013.

[5] R. Lyons, S. Macey, S. Turner, and D. Ford, "Development of a Record Linkage System To Support Injury Surveillance and the Evaluation of Interventions," *Injury Prevention*, vol. 18, no. Supplement 1, pp. A22–A23, Oct. 2012.

[6] A. Esscher, U. Högberg, B. Haglund, and B. Essën, "Maternal mortality in Sweden 1988-2007: more deaths than officially reported.," *Acta obstetricia et gynecologica Scandinavica*, vol. 92, no. 1, pp. 40–6, Jan. 2013.

[7] L. Gu, R. Baxter, D. Vickers, and C. Rainsford, "Record Linkage : Current Practice and Future Directions."

[8] I. P. Fellegi and A. B. Sunter, "A Theory for Record Linkage," *Journal of the American Statistical Association*, vol. 64, no. 328, pp. 1183–1210, 1969.

[9] T. Dalenius, "Finding a Needle In a Haystack or Identifying Anonymous Census Records," *Journal of Official Statistics*, vol. 2, no. 3, pp. 329–336, 1986.

[10] T. Churches, P. Christen, K. Lim, and J. X. Zhu, "Preparation of name and address data for record linkage using hidden Markov models," vol. 16, pp. 1–16, 2002.

[11] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[12] V. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” *Soviet physics doklady*, 1966.

[13] W. Winkler, “String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage.,” pp. 1184–1187, 1990.

[14] D. B. Dempster, A. P. and Laird, N. M. and Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[15] W. Winkler, “Using the EM algorithm for weight computation in the Fellegi-Sunter model of record linkage,” *Proceedings of the Section on Survey …*, 1988.

[16] M. A. Hernández and S. J. Stolfo, “The merge/purge problem for large databases,” *ACM SIGMOD Record*, vol. 24, no. 2, pp. 127–138, May 1995.

[17] M. A. Hernández and S. J. Stolfo, “Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem,” *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[18] W. Yancey, “Bigmatch: A Program for Large-Scale Record Linkage,” pp. 4652–4655, 2002.

[19] A. McCallum, K. Nigam, and L. H. Ungar, “Efficient clustering of high-dimensional data sets with application to reference matching,” *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’00*, pp. 169–178, 2000.

[20] H. Garcia-molina, “P-Swoosh : Parallel Algorithm for Generic Entity Resolution Hideki Kawai,” pp. 1–17.

[21] M. Yakout, M. J. Atallah, and A. Elmagarmid, “Efficient Private Record Linkage,” *2009 IEEE 25th International Conference on Data Engineering*, pp. 1283–1286, 2009.

[22] E. Durham, “A framework for accurate, efficient private record linkage,” 2012.

[23] R. Rivest, “On data banks and privacy homomorphisms,” *Foundations of secure …*, 1978.

[24] O. Benjelloun, H. Garcia-molina, and T. E. Larson, “Generic Entity Resolution in the SERF Project,” pp. 1–9.

[25] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, “Swoosh: a generic approach to entity resolution,” *The VLDB Journal*, vol. 18, no. 1, pp. 255–276, 2008.

[26] “match2lists.” [Online]. Available: http://www.match2lists.com/. [Accessed: 15-Oct-2012].

[27] "OpenEMPI: An Open Source Enterprise Master Patient Index." [Online]. Available: http://openempi.kenai.com/. [Accessed: 22-Oct-2012].

[28] L. M. Garshol, "duke: Fast deduplication engine." [Online]. Available: https://code.google.com/p/duke/. [Accessed: 13-Nov-2012].

[29] "Pentaho Data Integration." [Online]. Available: http://kettle.pentaho.com/. [Accessed: 03-Dec-2012].

# Appendix A – Kettle as a record linkage tool

Because of the interesting features provided by Kettle, our first proof-of-concept solution was created as an extension of Kettle. So, we attempted to use the provided functionality to perform record matching.

Firstly, a transformation would import two text files using plugins available in Kettle.

Then, the Join Rows plugin was used to produce the cartesian product of both files. In this case, we have files $A$ and $B$ with records $a_i = (\alpha_{i1}, \alpha_{i2} \dots \alpha_{in}), i \in [1, |A|]$ and $b_j = (\beta_{j1}, \beta_{j2} \dots \beta_{jn}), j \in [1, |B|]$ with $n$ being the number of fields for the records in each of the files. Because we are only concerned with matching, it is necessary that the record have the same structure, and therefore the same number of fields ($n$). The resulting set $A \times B = C$ has the records $c_{ij} = (\alpha_{i1}, \alpha_{i2} \dots \alpha_{in}, \beta_{j1}, \beta_{j2} \dots \beta_{jn})$, essentially creating a list of every pair of records. This means that all the information necessary to compare the two original records is contained in one single record, fitting in Kettle's row-oriented approach.

Next, a plugin that receives rows in this format was developed to be integrated with Kettle. Every row is split into the two records it contains and a matching function is applied to each field value. The plugin itself implements various matching functions, along with the weighting of the scores for each field. This plugin adds a score field to the row, with the score of the pair. To further generalize the process, the developed matching plugin could instead load various packaged matching functions, and allow the user to choose which function is applied to each field, and the weight of the score for each field. This was not developed however, as the basic concept had been proven.

To complete the process, a Filter Rows plugin can be used to redirect a row to a step depending on the value of the score. This step could, for example, merge two linked records or create a report with the possibly-linked records for clerical review.

This approach has several drawbacks, the first being that it cannot take advantage of the streaming nature of Kettle, because at least one of the record sets must be completely read at matching time, to produce the cartesian product set. In this case, the Join Rows plugin maintains the incoming rows in a temporary text files. Moreover, it would also require significant effort to produce a solution where a user could benefit from multiple matching functions. A truly generic approach would require a system to be built that could add user developed matching functions to this plugin. Because any temporary storage would need to be implemented by the step plugin, any method of blocking would also have to implement such storage.

Because of these difficulties, Kettle was not used as the basis of our work. However, once a tool fitting the presented requirements is developed, we envision a way to synergize with the Kettle. This can be done by developing a method to create a list of matches (or non-matches) that is then used by Kettle to process the records in question. This way, our framework would provide enough information for Kettle to choose if a given set of records match, and then Kettle would continue to apply the ETL process knowing this relationship.

# Appendix B – Plugin code

In this section we provide some code of the user-provided plugins developed. This also shows convenience of plugin development. This class shows a plugin to convert values to uppercase and removes accent and other symbols:

```
import java.text.Normalizer;
import java.util.HashMap;
import java.util.List;
import pt.ua.rlaas.data.Record;
import pt.ua.rlaas.data.Schema;
import pt.ua.rlaas.plugin.TransformPlugin;
import pt.ua.rlaas.util.RecordHelper;
public class ToUppercase implements TransformPlugin {
    public void init(HashMap<String, String> settings) {}
    public  List<Record>  transform(List<Record>  records,  Schema
schema, String[] fields) {
        RecordHelper rh = new RecordHelper(schema);
        for (Record r : records) {
            for (String f : fields) {
                String out = rh.get(r, f);
                out = out.toUpperCase();
                out          =          Normalizer.normalize(out,
Normalizer.Form.NFD);
                out = out.replaceAll("[^\\p{ASCII}]", "");
                rh.set(r, f, out);
            }
        }
        return records;
    }
    public void destroy() {}
}
```

The following class implements a plugin to measure the Levenshtein distance. The values are normalized by diving the distance by the greater length of the two words:

```
import java.util.HashMap;
import pt.ua.rlaas.plugin.ComparePlugin;
import pt.ua.rlaas.util.Util;

public class Levenshtein implements ComparePlugin {
    public void init(HashMap<String, String> settings) {}
    public double compare(String valueA, String valueB) {
        double lfd = computeLevenshteinDistance(valueA, valueB);
        double    score    =    Util.clamp(1.0d    -    (lfd    /
(Math.max(valueA.length(), valueB.length()))), 0.0, 1.0);
        return score;
    }

    public   static   int   computeLevenshteinDistance(CharSequence
str1, CharSequence str2) {
        int[][]    distance    =    new    int[str1.length()    +
1][str2.length() + 1];
        for (int i = 0; i <= str1.length(); i++) {
            distance[i][0] = i;
        }
        for (int j = 1; j <= str2.length(); j++) {
            distance[0][j] = j;
        }
        for (int i = 1; i <= str1.length(); i++) {
            for (int j = 1; j <= str2.length(); j++) {
                distance[i][j]  =  Math.min(Math.min(distance[i  -
1][j] + 1, distance[i][j - 1] + 1),
                        distance[i - 1][j - 1] + ((str1.charAt(i -
1) == str2.charAt(j - 1)) ? 0 : 1));
            }
        }
        return distance[str1.length()][str2.length()];
    }
    public void destroy() {}
}
```

This class selects the first three letters of a field and sets them as a block identifier:

```
import java.util.HashMap;
import java.util.List;
import pt.ua.rlaas.data.Record;
import pt.ua.rlaas.data.Schema;
import pt.ua.rlaas.plugin.TransformPlugin;
import pt.ua.rlaas.util.RecordHelper;

public class FirstLetterTaxonomy implements TransformPlugin {
    public void init(HashMap<String, String> settings) {
    }
    public  List<Record>  transform(List<Record>  records,  Schema
schema, String[] fields) {
        RecordHelper rh = new RecordHelper(schema);
        for (Record r : records) {
            String block = rh.get(r, field[0]).toUpperCase();
            if (block.length() >= 3) {
                block = block.substring(0, 3);
            }
            r.setBlock(block);
        }
        return records;
    }
    public void destroy() {
    }
```