A Software Infrastructure for the CLEENEX Optimizer

Telma Filipa de Carvalho Fernandes

Instituto Superior Técnico

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

Keywords: Approximate Duplicate Detection, Optimizer, Data Matching, Record Matching.

1 Introduction

Data cleaning is the process that aims at correcting data quality problems [15] occurring in a database. The existence of approximate duplicate records is a typical example of a data quality problem. A data cleaning process is, typically, modeled as a graph of data transformations. Data transformations are operations that we can apply to data. The detection of approximate duplicate records [2] [9] is one of the tasks of a data cleaning process. The purpose is to detect approximate duplicate records by determining whether two given records match or not in order to know if they refer to the same real-world entity. This process is also known as *data matching* [5].

There are some data cleaning tools that are specialized in correcting data quality problems. However, the main problem about most of the data cleaning tools is the fixed implementation for each logical operator. Due to this, it is not possible to choose the most appropriate approach to perform the approximate

duplicate detection. One solution to address this problem is to use an optimizer that can build several execution plans for the data cleaning program and, based on the cost of each execution plan, choose the most efficient. However, in order to have this optimizer, first we need to have an infrastructure in which the optimizer can be implemented.

We implemented an infrastructure to support an optimizer in CLEENEX [7], a data cleaning prototype. This infrastructure provides various algorithms that improve the efficiency of the *data matching* process, as an alternative to the *Cartesian Product*. Besides these algorithms to scale-up the *data matching* process, we also want to provide various string matching algorithms that allow the user to create rules that are better suited to each type of dataset. We performed the evaluation of the implemented algorithms to scale-up the *data matching* process in terms of efficiency and effectiveness. In these tests we obtained results with a gain in performance between 95% to 98% compared to the *Cartesian Product*. We also obtained results showing that in terms of effectiveness, these algorithms do not show significant differences comparing to the *Cartesian Product*.

The rest of this paper is organized in five sections. Section 2 explains record matching techniques to guarantee the effectiveness of the *data matching* process. In this section we also describe algorithms to improve the efficiency of *data matching* process. Section 3 presents CLEENEX, the data cleaning tool in which we implemented the infrastructure for the optimizer. Section 4 details the prototype implementing the infrastructure for the optimizer component in CLEENEX. Section 5 describes the validation methodology regarding the efficiency and effectiveness for each algorithm to improve the efficiency of the *data matching* process. Finally, Section 6 concludes and presents the future work of this thesis.

2 Related Work

The *data matching* process has two main challenges: effectiveness (i.e., accuracy) and efficiency. It is difficult to match a pair of records accurately because, typically, the records are not exact matches. This happens because records sometimes have typing errors, abbreviations, etc. *Record matching* techniques [8] address the problem of accurately determine whether a pair of records is a match.

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

However, this approach is not commonly used, since the results are not the more accurate. Instead, these string matching algorithms are used in *rule-based*

matching [4]. This approach classifies records into matches or non-matches by applying a set of rules. These rules are built with a set of similarity measures (e.g., string matching algorithms), that compare certain attributes whose values can help us determine whether the correspondent record belongs to the same real-world entity.

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

In order to efficiently match a very large amount of records, we need to minimize the number of record pairs to be compared. Some algorithms were proposed to reduce the number of comparisons, by avoiding to compare record pairs that do not refer to the same real-world entity. The following algorithms are examples of proposed solutions to improve the efficiency of the approximate duplicate detection process:

- The *Traditional Blocking* [10] is a simple technique that blocks the records according to a key value. A key is defined by the user and is formed with one or more attributes of the records. The records that share the same key value are inserted into the same block. Only the records that are in the same block are compared among each other.
- The *Sorted-Neighborhood Join (SNJ)* algorithm [13] was introduced to match similar records by sorting the records according to a key. This key is also formed with the values of one or more attributes. After the records are sorted, the algorithm moves sequentially a window with a fixed size over the sorted records. The sliding window goes through all the records and generate all the candidate records pairs until it reaches the end of the table. There are some other variants of the *SNJ* algorithm that are proposed improvements that overcome certain limitations of the traditional version of the *SNJ*.
- The *Q-gram Based Indexing* [3] assigns all records that have the same variation of their key into the same cluster. These variations are q-grams of the key values of each record that are reorganized in order to form other keys. The same record is then inserted into several clusters that correspond to each key value of the record's keys.
- The *Suffix Array Based Indexing* [1], which is very similar to the *Q-gram Based Indexing*. It also creates variations of the key, but it uses the key's suffix instead of using q-grams.
- The *Canopy Clustering* [6] [14] uses a computationally cheap similarity measure to group the records into overlapping clusters, also called *canopies*. Inside each canopy, the records are compared among each other with the use of another similarity measure. This similarity measure, normally, is more expensive than the one used to group the records.

3 Background

The CLEENEX [7] framework is a prototype for relational data cleaning that extends AJAX [11] [12]. AJAX is a data cleaning framework that enables the specification of data cleaning programs using a language that extends SQL.

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

At the logical level, CLEENEX supports five types of data transformations: **View**, an SQL query; **Map**, a one-to-many mapping between an input record and the corresponding output records; **Cluster**, that groups the records from an input relation according to a given clustering algorithm; **Merge**, that groups the records of an input relation, by a given criteria, and chooses a representative for each group; and **Match**, that applies an approximate join to two input relations. CLEENEX also supports the use of external functions (implemented in Java) to be invoked within transformations. All the functions are defined by the user and are stored in an external functions library. The external functions can be, for example, string matching algorithms that can be used by the match transformation.

At the physical level, certain decisions can be made in order to optimize the execution of a data cleaning program. More specifically, an efficient approach can be chosen to implement a given transformation. In particular, for the *Match* transformation, its naive implementation consists in the computation of a Cartesian product between the two input tables. For each candidate record pair it applies a similarity function in order to determine whether they can be considered as matches. The *Match* transformation is one of the most expensive transformation, specially if the two input tables contain a large amount of records.

The core operation of CLEENEX consists in the compilation phase and in the execution phase. In the compilation phase, the data cleaning program written by the user is analyzed by the parser component of CLEENEX. This analysis is important in order to guarantee that the program does not have any syntactical and semantical errors. After the correct parsing of the data cleaning program, the optimizer component of CLEENEX can choose the best physical implementation for each transformation. The *Optimizer* analyzes each data transformation and chooses if the execution of that transformation is either in Java or in SQL. After the *Optimizer* chooses the implementation for each transformation, a new Java class is created. CLEENEX writes in this class the correspondent code, based on existing templates for each type of transformation. After the code for each transformation is generated, all the created classes are then compiled in order to be executed. In the execution phase, the sequence of data transformations defined in the data cleaning program is executed in the same order as defined by the user. The corresponding Directed Acyclic Graph (DAG) that models the data cleaning program is displayed to the user through a Graphical User Interface (GUI).

4 Solution

In this section we present the realization of the infrastructure for the optimizer component of CLEENEX. In the current version of CLEENEX, the optimizer has a minor impact in the efficiency of the data cleaning program. The user starts by writing the data cleaning program into a file using a specification language that extends SQL. This file is given to the *Parser* in order for the program to be syntactically analyzed and parsed.

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

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

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

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

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

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

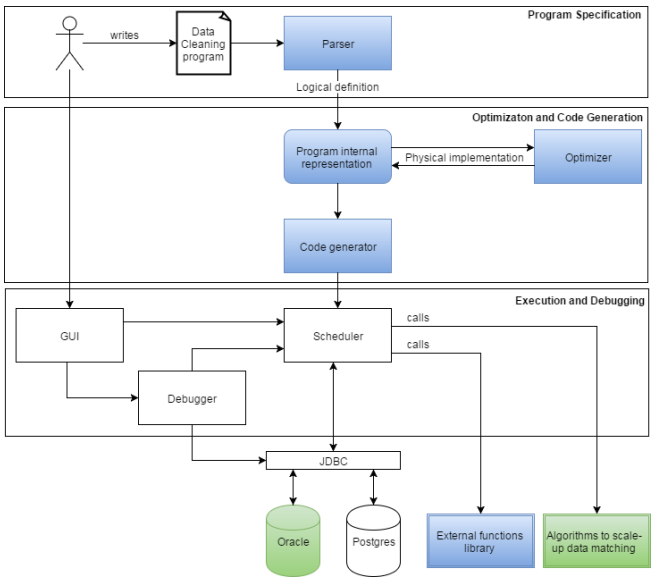

Figure 1: New architecture of CLEENEX

In Figure 1 we have the new architecture of CLEENEX, in which we highlight the current CLEENEX components that need modifications, namely the *Parser*, the *Optimizer*, the program internal representation, the code generator and the external functions library. We also added the new components that are were be inserted in the architecture, namely the algorithms to scale-up rule-based matching and the support to communicate with the Oracle Database.

5 Validation

We performed several experiments to validate the algorithm to scale-up *data matching* in terms of effectiveness (i.e., accurate results) and efficiency. We defined the datasets we used, the matching criteria (i.e., rules) we used to find approximate duplicates and the metric we evaluated. After defining several rules

for each dataset, we applied the *Cartesian Product* to know how many record pairs complied each rule. After we identified the rule that covered more record pairs for each dataset, we applied each algorithm to scale-up *data matching* with that same rule.

For each scale-up algorithm we defined different configurations (i.e., keys, window size, thresholds, etc) in order to verify how it would affect the results. In the end we gathered all the results and analyzed them in order to verify the behavior (in terms of effectiveness and efficiency) of each algorithm. We also took into account in how the chosen parameters of each algorithm caused a significant impact. For the effectiveness experiments we collected the number of matches and the number of false matches. With these values, along with the number of true duplicates (which we collected when we used the *Cartesian Product*), we computed the *precision*, the *recall* and the *f-measure* of each algorithm. For the efficiency experiments we executed 5 times each algorithm with the same configuration and collected the run-times and the number of comparisons (i.e., the number of candidate record pairs). With these values we calculated the minimum time, the maximum time and the average time.

In Figure 2 we have the overall effectiveness of the algorithms to scale-up *data matching*. For each dataset and for each algorithm we chose the algorithm with the best relation between the *f-measure* and the average run-time. We built this graph in order to compare all the scale-up matching algorithms for each dataset with the *Cartesian Product*. As we can see, in general, the effectiveness of the algorithms to scale-up data matching does not vary comparing to the *Cartesian Product*.

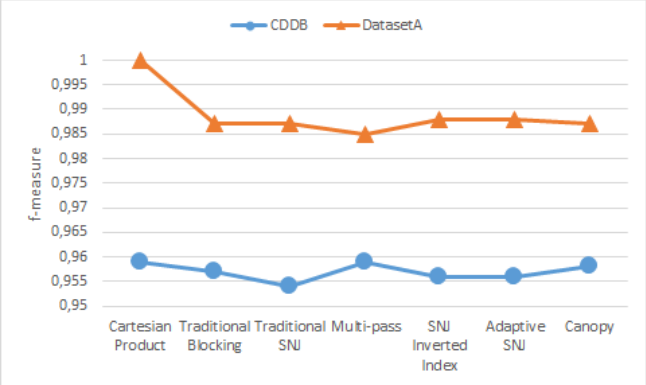

Figure 2: Overall results of the experimentations regarding effectiveness

In Figure 3 we have the overall efficiency of the algorithms to scale-up *data matching*. We used the average time of the same algorithms (and corresponding configurations) from the previous graph. We also collected the average time of the *Cartesian Product* and built a graph in order to compare the average times

(in milliseconds) of all approaches. As we can see, there is a noticeable difference in the average time values according to the algorithms to scale-up data matching.

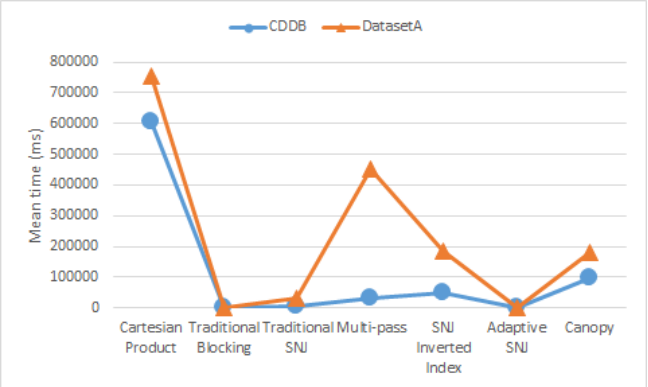

Figure 3: Overall results of the experimentations regarding efficiency

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

6 Conclusion

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

used to improve the efficiency of the data cleaning process. We also described the experimentations we made regarding the efficiency and effectiveness of each algorithm. We analyzed each result and concluded that there is a gain in terms of efficiency when using these algorithms to scale-up data matching instead of the *Cartesian Product*.

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

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

The current hint that allows to user to choose the key for the scale-up matching algorithm supports the definition of simple keys (i.e., keys that are formed with one attribute value) and composed keys (i.e., keys that are formed with parts of one or more attributes values). However, the syntax to define composed keys only supports the use of the first or last n characters of a given attribute value. The syntax of the composed keys could support more types of character extraction to form composed keys. For example, allow the use of functions that extracts all the numbers in a given attribute value to form the composed keys.

References

1. Akiko N. Aizawa and Keizo Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, 2005.
2. Mikhail Bilenko and Raymond J. Mooney. On evaluation and training-set construction for duplicate detection. In *KDD*, pages 7–12, 2003.
3. Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Inf.*, 56(1-2):51–70, 2003.
4. Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Data-Centric Systems and Applications. Springer, 2012.
5. William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, pages 73–78, 2003.

6. William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *ACM SIGKDD*, pages 475–480, 2002.
7. João Lobato dos Santos Dias. Support for user interaction in a data cleaning process. Master's thesis, Instituto Superior Técnico, 2012.
8. AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.
9. Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.
10. Ivan P. Fellegi and Alan B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.
11. Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.
12. Helena Galhardas, Antónia Lopes, and Emanuel Santos. Support for user involvement in data cleaning. In *DaWaK*, pages 136–151, 2011.
13. Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD*, pages 127–138, 1995.
14. Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *ACM SIGKDD*, pages 169–178, 2000.
15. Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE DEB*, 23(4):3–13, 2000.