# Integrating Approximate Duplicate Detection into Pentaho Data Integration

Sílvia Castro Fernandes
Instituto Superior Técnico

## ABSTRACT

Data has increasingly become a fundamental asset for the great majority of companies and it keeps being produced at a very high rate. Large datasets are prone to data quality problems that can have a negative impact on the results of the analysis based on the data. The presence of approximate duplicate records is a data quality problem that arises commonly in a data integration context. Data integration consists in a set of processes that enable the integration of different data sources.

Approximate duplicate records are records that represent the same real-world entity. If this problem is not detected, these records are processed as separate entities, influencing the results of analysis performed on the data. For example, if a client is present in two different data sources that are integrated, that client's record may be duplicated in the final data source and the client will be treated as two different clients. Data integration tools can benefit from an approximate duplicate detection mechanism, but not all tools provide one. Pentaho Data Integration (PDI) is an open source data integration tool that was used as a case study for the inclusion of approximate duplicate detection.

This work proposes two PDI steps that allow the user to compute approximate duplicates. These steps compute groups where all the elements of each group are considered approximate duplicates that represent the same real-world entity. There were also modifications made to an already existing PDI visualization to facilitate the visualization of the steps results. Our solution was evaluated considering correctness, performance and usability. The results of the evaluation were overall positive, offering users the possibility to compute approximate duplicate records without having to perform cartesian products of the data and without the need to be highly familiar with PDI.

## KEYWORDS

Approximate Duplicate Detection, Data Profiling, Data Quality, Pentaho Data Integration

## 1 INTRODUCTION

Data has increasingly become a fundamental asset for the great majority of companies, being them IT related or not. Big volumes of data must be handled and, sometimes, the people responsible for processing the data may have close to no information about the dataset. These large datasets usually present data quality problems, like missing or inconsistent data. Evaluating the quality of a dataset is relevant because decisions will be made based on that data and if it is not clean and consistent, those decisions will not be reliable. To improve data quality, it is important to collect data about the data, designated *metadata*, which may include the names and types of the attributes, but also properties of the data like the number of null records, value distributions or dependencies between attributes.

*Data profiling* [1] is the activity that encloses a set of processes to obtain metadata regarding the dataset that is being handled. Data profiling has several applications, one of them being data integration. *Data integration* [3] is defined as the set of techniques that enable a uniform access to a set of autonomous and heterogeneous data sources through a common schema, enabling to query that common schema in order to get results from the various datasets simultaneously. Data integration is prone to encounter data quality problems, in particular, the occurrence of approximate duplicate records, meaning records that may not be identical but represent the same real-world entity. For example, if a costumer is present in two different records of a company database, we want to detect these records to prevent keeping and processing them as different entities.

There are several data integration tools in the market, but not all provide data profiling capabilities. In particular, Pentaho Data Integration (PDI) is an open source tool that has limited data profiling functionality. PDI is an Extract, Transform and Load (ETL) tool that enables to integrate different types of data sources, apply some operations on the data, like sorting or merging data, and then save the results in a specified format. These operations are represented by steps. *Steps* are connected by hops. *Transformations* are a set of steps and hops. An example of a PDI transformation is shown on Figure 1. We use PDI as a case study for addressing the limitations

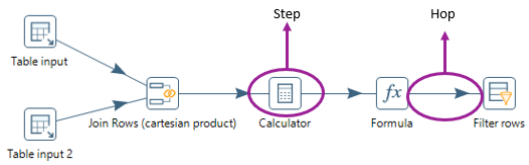


**Figure 1: A PDI transformation**

of data integration tools in terms of data profiling functionality, focusing on approximate duplicate detection. Since approximate duplicate detection is an important task to ensure data quality when integrating different data sources, it is a relevant task to be supported by data integration tools such as PDI. It is already possible to create a workflow to detect approximate duplicate records in PDI, like the transformation in Figure 1. It joins the data from two tables, performing a cartesian product of the data. Then, attribute similarity is calculated between records and a formula to specify how the similarities should be added is defined. Finally, the rows that do not satisfy the rule defined in the previous step are filtered out. However, this approach has several limitations: i) since we perform a cartesian product of the data, this will raise efficiency problems, specially for large datasets; ii) it may not be intuitive for a user that is not familiar with PDI to build a transformation to

perform this task, requiring that the users are familiarized with a range of steps and their configurations.

The main objective of this work is to extend PDI's capabilities to support data profiling, focusing on the implementation of a flexible and efficient mechanism to detect approximate duplicate records, that users can use even if they are not very familiar with PDI or the approximate duplicate detection domain.

To achieve our goal, we implemented two PDI steps and modified a visualization to help interpret the steps results. The first step to be developed, the Domain-Independent Duplicate Detection step, enables the user to compute approximate duplicate groups without the need to provide domain-specific information regarding the dataset. The other step, the Approximate Duplicate Detection step, allows to assign weights and specific measures to compute the similarity between the values of each field in the dataset to be used in computing approximate duplicate groups. Both steps provide alternatives to performing a cartesian product of the data. A Table visualization that already existed in PDI was modified to make it easier for users to visualize groups of records. The steps were evaluated for correctness and performance, and user tests were performed to evaluate usability. The visualization was also submitted to user testing.

The document is organized as follows: Section 2 describes algorithms and strategies used in the context of approximate duplicate detection. Section 3 describes PDI and its main concepts, and presents the work done to integrate new ways of performing approximate duplicate detection in PDI, as well as the modifications made to a visualization to help interpret the steps results. In Section 4, the evaluation of the work regarding correctness, performance and usability is described. In Section 5, we present our conclusions and future work.

## 2 BACKGROUND

When integrating different data sources, usually there is not a universal identifier for the records in the context of both data sources. Therefore, when data is integrated it is possible that we are left with more than one record corresponding to the same real-world entity. The main goal of approximate duplicate detection is to identify different records that refer to the same real-world entity. Approximate duplicate detection can be considered as a data profiling task, since knowing which records are approximate duplicates is information we can gather from the data. Approximate duplicate detection can be performed by applying field matching techniques to individual attributes. However, in most real-life cases, since records are composed of several attributes, we want to analyze more than one attribute to decide if different records correspond to the same real-world entity. For this, we use record matching techniques to obtain the approximate duplicate records. Since approximate duplicate detection may lead to a very large number of comparisons, efficiency problems soon arise. Therefore, it is important to use techniques that allow to reduce the overhead of this process.

There are several record-matching techniques [3], that include rule-based matching, learning-based matching, matching by clustering and matching based in probabilistic approaches. We describe in more detail rule-based matching, since it was implemented in one of our steps.

There are several strategies that can be used to reduce the large number of comparisons that is inherent to comparing all records with each other. Such techniques include hashing, blocking, the sorted-neighbourhood method, clustering and canopies. We describe blocking, that was implemented in the step that performs rule-based matching and present an approach proposed by Monge and Elkan to compute approximate duplicate groups in a domain-independent way, also reducing the number of comparisons between records.

### 2.1 Rule-based matching

*Rule-based matching* consists in defining rules that combine the calculations of similarity metrics for several fields and give each field similarity a specific weight. For example, to match records from two data sources that have the attributes *name* and *phone*, we can define a rule stating that two records are approximate duplicates if the corresponding similarity is greater than 0.8, where the similarity between names would have a weight of 0.6 and the similarity between phone numbers would have a weight of 0.4: *sim(x, y) = 0.6*simName(x, y) + 0.4*simPhone(x, y)*. When defining a linearly weighted combination of the individual scores, as the example provided above, the sum of the weights should add to 1. Rules may be more complex and are often written using a high-level declarative language, but they can become hard to write, the weights of the different attributes may be hard to set and sometimes it is not clear how the rules should be specified.

### 2.2 Blocking

*Blocking* [4] is a technique used to reduce the number of record comparisons when we are detecting approximate duplicates. It consists in partitioning the data in such a way that the records that have the same value for one or more specified attributes are included in the same block. The blocks are mutually exclusive, meaning that one record can only belong to one block. The comparisons are only performed between records that belong to the same block.

### 2.3 An efficient domain-independent algorithm for detecting approximately duplicate database records

This work proposed by Monge and Elkan [6] consists in an approach that allows to compute approximate duplicate groups without the need to provide domain-specific information. This approach also presents an alternative to comparing all records to each other. The base idea of the algorithm is to concatenate all the fields of a record into a single string and compare the several strings using a similarity measure to create clusters that represent the same real-world entity.

To reduce the number of comparisons, the concept of transitivity (if $a$ is an approximate duplicate of $b$ and $b$ is an approximate duplicate of $c$, then $a$ is an approximate duplicate of $c$) is applied. With this, the approximate duplicate detection problem can be modeled as a problem of finding connected components in an undirected graph, where each connected component corresponds to a group of records that represent the same real-world entity. For this, a record is represented by a node in the graph and an edge between two nodes represents the "is approximate duplicate of" relation. This way, if $a$ and $b$ are approximate duplicates, there will be an edge

connecting them. This approach helps to reduce the number of necessary comparisons, since if two records belong to the same connected component, they are already considered approximate duplicates and there is no need for them to be compared to each other. If two records are not in the same component, then we must compare them and, if they match, we add an edge between them, creating a new connected component. To model this problem, a union-find structure [2] is used. This structure keeps a collection of disjoint sets where each set has a representative. This structure has two essential operations:

- *Union(x, y)*: This operation over the nodes $x$ and $y$ merges the components that contain $x$ and $y$ into a new component that will replace the previous two in the collection.
- *Find(x)*: This operation returns the representative of the component that contains node $x$. If $x$ and $y$ belong to the same component, then Find($x$) = Find($y$).

This algorithm performs two passes over the data. For the first pass, the strings that were created by concatenating the fields of each record are ordered lexicographically from left to right and, for the second pass, the strings are ordered lexicographically from right to left. To further reduce the number of necessary comparisons, a priority queue is used. This priority queue keeps sets of records that will be used to compare to the remaining records. It keeps up to four record sets, where each set corresponds to the same real-world entity. These sets will represent the most recently found clusters. The processing of the records is similar for both passes. As data is processed, we want to compute the group for record $a$. Record $a$ is compared to the records in the queue. First, we compute Find($a$) and verify if its representative is present in the priority queue. If it is present, then no more comparisons are needed for that record and the corresponding cluster will receive maximum priority in the queue. This verification can be ignored in the first pass of the algorithm, since the records that are being processed were not yet grouped. If $a$'s representative is not present in the queue, we use a similarity metric to compare $a$ to the records present in the queue. The used similarity measure is the Needleman-Wunsch measure [7]. When a match is found, a Union operation is performed between record $a$ and the record that matched, but the cluster is not given maximum priority in the queue. If $a$ has a similarity value greater that the matching threshold but below a certain inclusion threshold, it may be included in the queue's set corresponding to the cluster it has joined and also be used for comparisons. If record $a$ does not match any of the records present in the queue, it is added to the queue as a singleton group with maximum priority. If an insertion results in the priority queue exceeding its capacity, the set corresponding to the cluster with the less priority is removed.

# 3 APPROXIMATE DUPLICATE DETECTION STEPS

In this Section, we start by describing the main concepts of PDI and a PDI step architecture (Section 3.1). Then, we describe the steps developed to extend PDI's capabilities to detect approximate duplicates. We describe first the Domain-Independent Duplicate Detection step (Section 3.2) and then the Approximate Duplicate Detection Step (Section 3.3).

## 3.1 Pentaho Data Integration

PDI is an ETL tool by Hitachi Vantara. It uses a drag-and-drop approach to perform data transformations. We can create transformations and jobs to perform data integration tasks. A *step* is a configurable building block of a transformation that performs a specific task, such as reading data from a file or sorting the dataset by a given attribute. A *hop* is a data pathway that connects steps and allows metadata to flow in the transformation. A *transformation* is a workflow composed of steps that are connected by hops. PDI deals with data streams so, for most steps, the data is processed sequentially and, as soon as a step finishes processing a row, that row can go into the next step, not having to wait for the whole dataset to be processed by a step before progressing to another.

Regarding approximate duplicate detection, it is possible do perform this task by building a transformation such as the one illustrated in Figure 1. To compare values of a specific field, PDI provides the following string matching techniques: edit distance (Levenshtein distance), Metaphone, double Metaphone, Damerau-Levenshtein distance, Needleman-Wunsch, Jaro, Jaro-Winkler, Soundex and refined Soundex.

PDI's enterprise edition provides the Data Exploration Tool (DET) that supports the visual inspection of the data that is being processed in a transformation through a set of visualizations, providing a better understanding of the data. The available visualizations are tables, bar charts, column charts, line charts, area charts, pie charts, doughnut charts, sunburst charts, scatterplots, bubble charts, heat grids and geographical maps.

*3.1.1 PDI Step Architecture.* A PDI step operates on a stream of data rows. These data rows are composed of the data that is being processed and the metadata associated to it, for example the parameters of the step. A step has a life cycle, represented in Figure 2, that consists of three stages: initialization, row processing and clean-up.

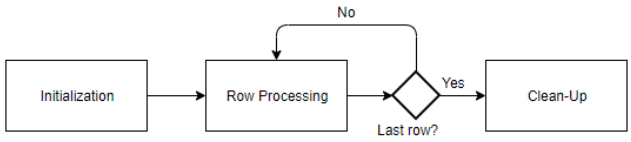


**Figure 2: A PDI step life cycle**

- *Initialization*: This phase corresponds to performing one-time initialization tasks, such as opening files and establishing database connections.
- *Row processing*: When a step starts receiving input, it enters a loop to process the input rows. For most steps, one row is processed and its structure and fields may be changed (for example, one field may be added to the row), then it is sent to the next step before another input row starts being processed. However, this kind of processing does not occur for some steps, such as steps that sort the input, that need to keep already processed rows for further processing (for example, sorting all the rows), before sending them to the

next step. The row processing stage finishes when there are no more input rows to process.
- *Clean-up*: When a transformation finishes, meaning there is no more data to be processed, all steps in the transformation must release all previously allocated resources and ensure that all opened files and database connections are closed.

## 3.2 Domain-Independent Duplicate Detection step

The algorithm that we implemented for the Domain-Independent Duplicate Detection step is described in Section 2.3 and was proposed by Monge and Elkan[6]. This algorithm computes approximate duplicate clusters, from now on mentioned as *groups*, without requiring any domain-specific information.

*3.2.1 Step Logic.* The Domain-Independent Duplicate Detection step receives input from a previous step in a transformation and outputs the input data plus two additional columns. The first column corresponds to the identifier of the group to which a record belongs. The group identifier corresponds to the index of the representative record (this index corresponds to the initial position of the record in the input data), where the representative record is the record that appears first in the data (for example, if one group has records with indexes 3, 4 and 5, the representative will be record 3 and, consequently, the group identifier will be 3). The second column corresponds to the similarity between a record and the representative of its group. For the representative of one group with more than one element, the similarity is null while the similarity value for the representative of a group that only contains one record is -1. This distinction is made to help the users better distinguishing the records that were not grouped with any others.

Our implementation presents three modifications from the original algorithm. The first modification is that our priority queue only keeps the representative records for each group. This occurs because the similarity threshold is a parameter configurable by the user. The inclusion threshold should be a value above the similarity threshold that would add more records of one group for comparisons. But since the similarity threshold is not a fixed value, we could not define a global value to create the inclusion threshold. If it was too high, no records would be added to the queue, if it was too small, the risk of obtaining a large number of false positive matches would increase.

The second modification consists of performing an additional verification before the Union operation: the Union operation is only applied to two groups if all the elements of the first group meet the similarity threshold with the representative of the second group and vice-versa. This modification represents an increase in precision values for the algorithm, meaning that there are less records being wrongly assigned to the groups to which they do not belong. It is also important to ensure that the users see similarity values that correspond to their expectations, meaning that if a user defines a threshold of 0.5 and this verification is not made, the output of the step may reveal similarity values below 0.5. This happens because the algorithm takes advantage of transitivity, where if record $a$ and record $b$ are approximate duplicates and record $b$ and record $c$ are approximate duplicates, then $a$ and $c$ are also approximate duplicates. This means that in one group, there may be records that do not meet the similarity threshold with the remaining elements of the group. Although our modification weakens the advantages of transitivity, it enables the step to output more coherent results to the users, since the outputted similarity values will all meet the threshold the user defined.

The third modification to the original algorithm corresponds to not considering a record that was not matched with any of the records in the queue in the second pass of the algorithm as a new singleton group. This means that, when the record is compared to all the records in the queue and no match is found, we add to the queue the representative of the record group instead of adding the record as a singleton group. This modification corresponds to an increase in the recall values, meaning that more records are correctly assigned to their groups.

Excluding these modifications, the step follows the flow of the original algorithm. As the input records are processed, the strings of the concatenated fields are created. When this is done for all records, the records are ordered based on the lexicographic order of those strings from left to right. Then, the record comparisons are performed. When all the records have been compared, the records are ordered based on the lexicographic order of the string from right to left and a similar processing to the first pass is performed. When all the records are processed, the step output is assembled.

*3.2.2 Step UI.* The Domain-Independent Duplicate Detection step UI is defined using the SWT toolkit[1] that is, by default, used in PDI. The Domain-Independent Duplicate Detection step has a simple UI. As illustrated in Figure 3 , it contains the following configurable parameters:

- *Step Name*: The name that will appear in the transformation. It does not affect the step execution or output.
- *Similarity Threshold*: The matching threshold used to verify whether two records are considered approximate duplicates. It is, by default, 0.5.
- *Grouping Column Name*: The name of the output column that contains the identifier of the group to which a record belongs. It is, by default, Group.
- *Similarity Column Name*: The name of the output column that contains the similarity computed between a record and the representative of the group it belongs to. It is, by default, Similarity.
- *Remove Duplicates*: When selected, only the representative of each group is included in the step output.
- *Remove Non-Duplicates*: When selected, the records that are alone in a group are removed from the step output.

The user can also use the *Help* button to access the step documentation which provides a brief explanation of what the step does and each of the parameters.

## 3.3 Approximate Duplicate Detection step

The Approximate Duplicate Detection step enables the user to perform rule-based matching, as described in Section 2.1, where the user can specify which attributes are used to compute similarity

[1] https://www.eclipse.org/swt/

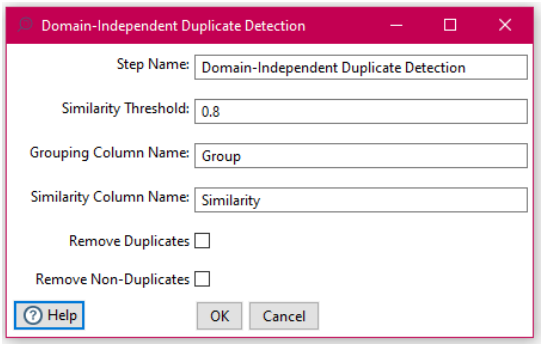


**Figure 3: Domain-Independent Duplicate Detection step UI**

between two records as well as the measures to compute the similarity between the values of each field and the weight it should have in the overall similarity.

*3.3.1 Step Logic.* The Approximate Duplicate Detection step receives as input the data outputted by a previous step in a transformation and outputs the input data plus two additional columns. The first column corresponds to the identifier of the group the record belongs to, where the identifier corresponds to the index of the first element of the group. In the step context, the index of a record is its position is the input dataset. This means that if the group contains the records that were first, second and third in the input data, the group identifier will be 1. The second column corresponds to the average similarity of the group the record belongs to. The Approximate Duplicate Detection step has different processing stages to compute approximate duplicate groups, that are illustrated in Figure 4.

First, the input records are partitioned into blocks using the technique described in Section 2.2. Then, for each block, the similarity between the records of the block is computed, based on the configurations provided by the user. The user chooses which attributes should be used to compute similarity, which similarity algorithm to use to compute the similarity between records of each field and the weight it will have in the overall similarity value. The threshold that the similarity between two records records must meet for them to be considered approximate duplicates is also defined by the user.

After the comparisons within the block are performed, it is needed to compute the approximate duplicate groups. Groups are formed in a way that a group is created for each record, where the group elements are the record and all the records whose similarity value meets the similarity threshold. For example, if records 3, 4 and 5 meet the similarity threshold with record 1, a group consisting of records 1, 3, 4 and 5 will be created. After the groups are created, it is possible that one record belongs to more than one group. We need to ensure that one record only belongs to one group in the end of the processing. To do this, we iterate through the elements of each group and verify if they belong to other groups. If a record belongs to two groups, but one of the groups includes the other group, the smaller group can be discarded since the records are already considered approximate duplicates in the larger group. For example, if one group contains records 1, 2, 3, and 4, and the other group contains records 2 and 3, the second group will be discarded,

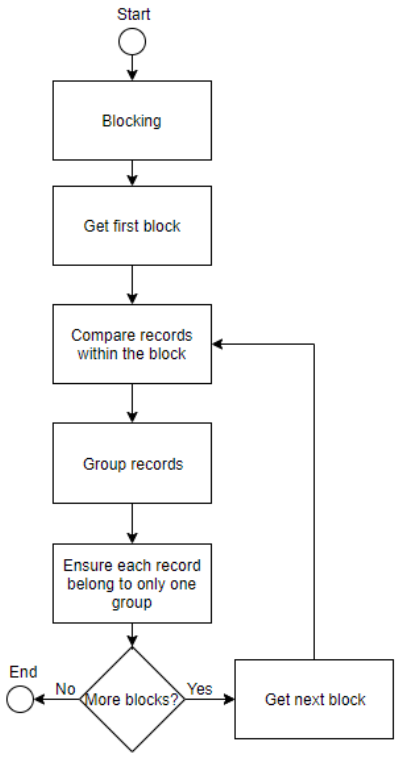


**Figure 4: Flow chart of the Approximate Duplicate Detection step processing**

since records 2 and 3 are already approximate duplicates in the first group. If this does not happen, the record that belongs to two groups will have to be removed from one of the groups. We remove the record from the group where it less affects the overall similarity. This is decided by computing the average similarity between the record and the remaining elements of the group for both groups and comparing those values. If one record has an average similarity of 0.4 in one group and an average similarity of 0.6 in another group, the record will be removed from the first group. This operation is performed until every record only belongs to one group. When this processing is done for the records in all blocks, the output of the step is assembled.

*3.3.2 Step UI.* The Approximate Duplicate Detection step has more configuration options than the Domain-Independent Duplicate Detection step, which leads to a UI with more elements. The step UI is divided by sections, as illustrated in Figure 5. First, there is the *Step Name* option. This changes the name of the step in the transformation but does not affect the step execution or output.

In the *Partitioning Attributes* section, there is a table that allows the user to select attributes to partition the data using a blocking approach. If no attributes are selected, a cartesian product of the data is performed.

In the *Matching Rules* section, the *Similarity Threshold* parameter corresponds to the threshold that the overall similarity between two records should meet so that they can be considered approximate

duplicates. In the *Fields* table, the user must add the attributes to use for computing the similarity. There are three columns to fill in:

- *Name*: The name of a field to be used in the computation of the overall similarity.
- *Measure*: The similarity measure to use for computing the similarity between values for that field.
- *Weight*: The weight a field similarity should have in the overall similarity. If the sum of weights does not add up to one, a warning will be shown to the user.

The *Get Fields* button automatically fills in the first column of the Fields table with all the attributes present in the data.

In the *Output* section, the parameters are the same as in the Domain-Independent Duplicate Detection step.

- *Grouping Column Name*: The name of the output column that contains the identifier of the group to which a record belongs. It is, by default, Group.
- *Similarity Column Name*: The name of the output column that contains the average similarity of the group to which the record belongs. It is, by default, Similarity.
- *Remove Duplicates*: When selected, only the first element of each group is added to the output of the step.
- *Remove Non-Duplicates*: When selected, the records that belong to a group with no more elements are removed from the step output.

As with the previous step, the *Help* button allows access to the step documentation which provides a brief explanation of what the step does and each of the parameters.

## 3.4 Table Visualization

To help the users interpreting the steps results, we modified the already existing Table visualization in DET. The Table visualization allows the user to see the output data of a step, enabling to choose which attributes should be displayed in the table through the Columns visual role. A *visual role* consists in a way for the user to specify how fields should be shown in the visualization. The user can also order the data by a specific attribute.

In order to facilitate the visualization of groups of records within the data, we created a new visual role designated *Group By*. This visual role accepts fields of any type, but only accepts a single field. When a user drags a field to that role, that field is brought to the first column of the table, the data is ordered based on its values and the table rows are colored according to each value of the field. This will result in records that have the same value for that field having the same colour, so it will be easier to visually identify the grouping. This is specially useful for when the dataset has a large number of fields that span beyond the width of the screen. We can still see which attributes are in the same group, since they will have the same color. We defined a color palette of 6 colors. This means that every 6 groups, the color of the group will be repeated. We chose colors that are easily distinguishable when being close to each other, specially the colors that are directly together. For example, if we used the Approximate Duplicate Detection step to detect approximate duplicates in a dataset about restaurants and we want to see how many restaurants there are based on the type of cuisine, we can drag the field that contains the data regarding the type of cuisine, in this case the *type* field, to the *Group By*

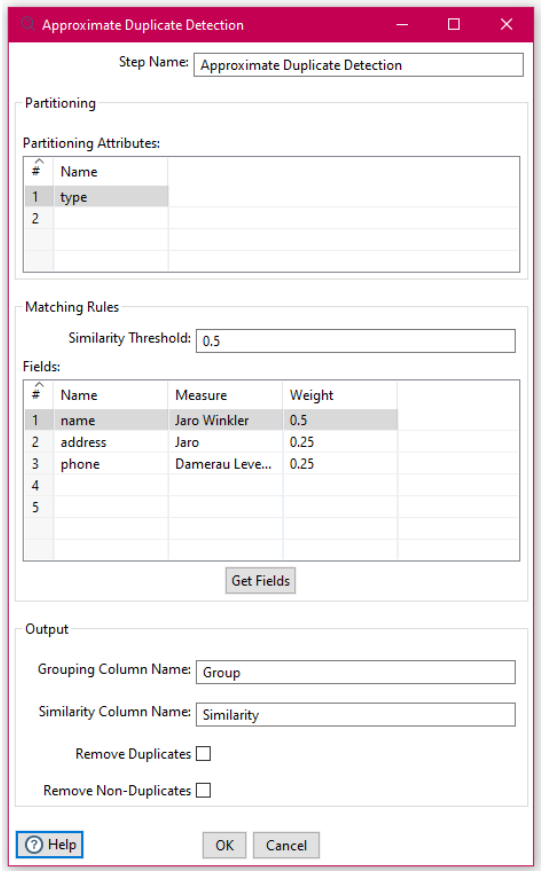


**Figure 5: Approximate Duplicate Detection step UI**

visual role. Then we can visualize all the restaurants with a specific type, for example *italian*, together and with the same color, which will facilitate our inspection of the relevant data to our task. The modified Table visualization is illustrated in Figure 6.

| Type | Name | City | Group | Similarity |
|---|---|---|---|---|
| italian | carmine's | new york | 57 | 0,9 |
| italian | carmine's | new york city | 57 | 0,9 |
| italian | felidia | new york | 67 | 0,9 |
| italian | felidia | new york city | 67 | 0,9 |
| japanese | katsu | los feliz | 21 | 0,7 |
| mexican | mi cocina | new york | 97 | 0,9 |
| mexican | mi cocina | new york city | 97 | 0,9 |
| nuova cuc... | rex il ristorante | los angeles | 39 | 0,7 |
| pacific ne... | chinois on main | santa monica | 11 | 0,9 |
| scandinavi... | aquavit | new york city | 49 | 0,9 |
| seafood | matsuhisa | beverly hills | 29 | 0,9 |
| seafood | le bernardin | new york city | 82 | 0,9 |
| seafood | manhattan ocean club | new york | 91 | 0,9 |
| seafood | manhattan ocean club | new york city | 91 | 0,9 |

**Figure 6: The modified Table visualization**

## 4 EVALUATION

Our solution was evaluated regarding correctness (Section 4.1), performance (Section 4.2) and usability (Section 4.3).

### 4.1 Correctness

Both steps were evaluated regarding correctness. They were evaluated using 5 datasets, one with information regarding restaurants[2], the CORA dataset[3] that contains bibliographic data, and the datasets from FEBRL[4]. For all these datasets, we collected precision and recall values and computed the F1-score, that corresponds to *2 * (precision * recall) / (precision + recall)*.

Since the output of the steps is a set of groups, we collected these values considering decisions over the N * (N + 1) / 2 pairs of records [5]:

- True Positive: two approximate duplicate records are assigned to the same group.
- False Positive: two not approximate duplicate records are assigned to the same group.
- False Negative: two approximate duplicate records are not assigned to the same group.

This allows to calculate precision as *true positives / (true positives + false positives)* and to calculate recall as *true positives / (true positives + false negatives)*.

For the Domain-Independent Duplicate Detection step, we computed precision, recall and the F1-score for the different modifications we performed on the original algorithm, for five different similarity thresholds: 0.4, 0.5, 0.6, 0.7 and 0.8. Since the results of the FEBRL datasets are similar, we present in Figure 7 the obtained results for the restaurants dataset, the CORA dataset and one of the FEBRL datasets.

Our implementation led to an overall increase in the F1-score, presenting positives results for most of the datasets. The obtained F1-scores for the CORA dataset in our implementation are worse than the values for the original algorithm, but this dataset has a large number of null values, so we still considered the modifications relevant, since the implementation yielded better results for the remaining datasets.

Regarding the Approximate Duplicate Detection step, the correctness of the results is highly dependent of the configurations performed by the users, so we computed precision, recall and F1-score for example configurations. For the restaurants, CORA and FEBRL1 datasets, we computed these values three times: one performing the cartesian product and two performing blocking with different attributes. The the remaining FEBRL datasets, we did not perform the cartesian product, because the step presented performance problems, as described in Section 4.2. The obtained results for the restaurants dataset are presented in Table 1, for the CORA dataset in Table 2 and for the FEBRL datasets in Table 3.

We can observe that, for most cases, the F1-score decreases when using blocking attributes. This occurs because blocking partitions the data in such a way that if two approximate duplicate records do not have the same value for the blocking attribute, they will not be compared. Still, when the decrease in the F1-score is small, it

[2]https://hpi.de/naumann/projects/repeatability/datasets/restaurants-dataset.html
[3]https://hpi.de/naumann/projects/repeatability/datasets/cora-dataset.html
[4]https://recordlinkage.readthedocs.io/en/latest/ref-datasets.html

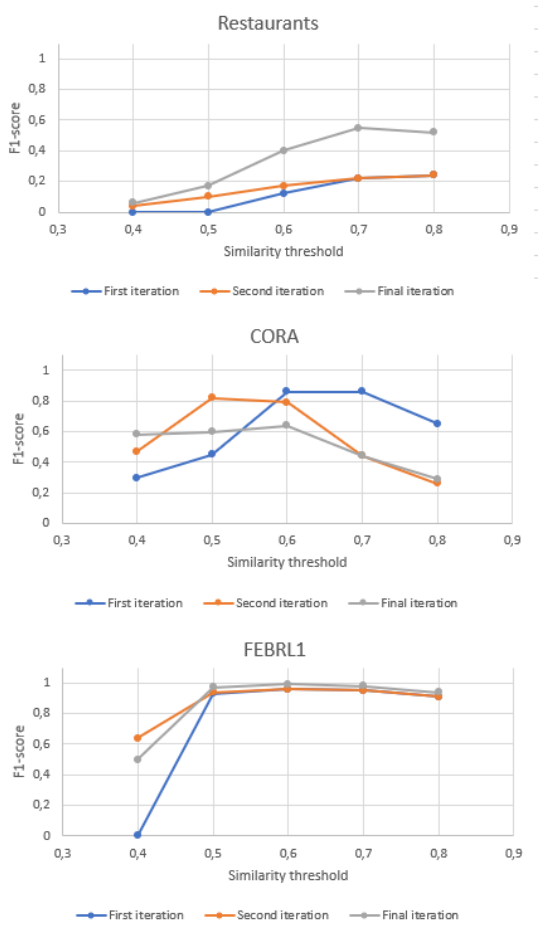


**Figure 7: F1-score evolution for different datasets**

| | P | R | F1 |
|---|---|---|---|
| **Cartesian Product** | 0.92 | 0.96 | 0.94 |
| **Blocking with *city*** | 0.86 | 0.54 | 0.66 |
| **Blocking with *type*** | 0.91 | 0.19 | 0.31 |

**Table 1: Precision, recall and F1-score for the Approximate Duplicate Detection step with the restaurants dataset.**

| | P | R | F1 |
|---|---|---|---|
| **Cartesian Product** | 0.78 | 0.34 | 0.47 |
| **Blocking with *institution*** | 0.89 | 0.31 | 0.46 |
| **Blocking with *journal*** | 0.83 | 0.41 | 0.55 |

**Table 2: Precision, recall and F1-score for the Approximate Duplicate Detection step with the CORA dataset.**

means that the chosen attribute is mostly correct throughout the dataset. In the CORA dataset, we can still observe that the *journal* attribute is a good partitioning attribute, yielding better results than the cartesian product. This means than the *journal* attribute groups together records that are approximate duplicates that are

| | | P | R | F1 |
|---|---|---|---|---|
| **FEBRL1** | **Cartesian Product** | 0.99 | 0.97 | 0.97 |
| | **Blocking with *suburb*** | 1.00 | 0.70 | 0.82 |
| | **Blocking with *state*** | 0.99 | 0.91 | 0.95 |
| **FEBRL2** | **Blocking with *suburb*** | 0.98 | 0.60 | 0.74 |
| | **Blocking with *state*** | 0.95 | 0.86 | 0.90 |
| **FEBRL3** | **Blocking with *suburb*** | 1.00 | 0.58 | 0.73 |
| | **Blocking with *state*** | 1.00 | 0.85 | 0.92 |

**Table 3: Precision, recall and F1-score for the Approximate Duplicate Detection step with the FEBRL datasets.**

not detected in the cartesian product, which may happen because of other records in the dataset that have high similarity in the *authors* and *title* attributes but differ in other attributes.

*4.1.1 Discussion.* Based on the obtained results, we can infer that both steps yield positive results. The Domain-Duplicate Detection step performs well, performing better when the fields in the first and last positions in the dataset present values that have a high confidence of being correct. Its results become less satisfactory when there is a large number of null values, as it is visible by the obtained results for the CORA dataset. Regarding the Approximate Duplicate Detection step, although its results are always dependent of the configurations provided by the user, it yields positive results if well configured. Although the step presents a scalability problem when the cartesian product is performed, as further discussed in Section 5.3.2, it is able to correctly detect most approximate duplicate groups. When using the blocking strategy, the step identifies the majority of approximate duplicates within each block, but presents lower recall levels is the cases where the field used to partition the data is not highly discriminating. Even for the CORA dataset, when performing blocking, we obtained acceptable results, since all the records that have a null value for the partitioning attribute are placed in the same block, so approximate duplicate records that have a null value for that field may be grouped together.

## 4.2 Performance

To evaluate the performance of the developed steps, we generated datasets using the FEBRL generator[5]. We generated datasets of 1000, 2000, 3000, 4000, 5000, 10000, 20000, 30000, 40000 and 50000 records where each presented, at maximum, 5 approximate duplicates per original record, leading to groups of at most 6 elements. We generated additional 15 datasets: 5 datasets of 1000 records, 5 datasets of 10000 records and 5 datasets of 50000 records, where each of the five datasets presented, at maximum, 10, 20, 30, 40 and 50 records per original record.

For the Domain-Independent Duplicate Detection step, we collected the execution time of the step using thresholds of 0.4, 0.5, 0.6, 0.7 and 0.8. Then, we calculated the average execution time for each dataset. The Domain-Independent Duplicate Detection step presented a linear growth of the execution time as the number of records increases, as illustrated in Figure 8. The increase of the number of records per group does not have a significant impact in the execution time of step, as illustrated in Figure 9.

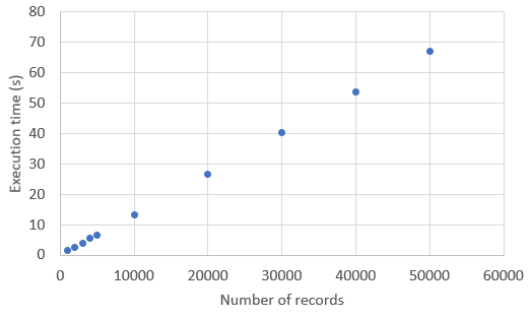


**Figure 8: Average execution time per number of records for the Domain-Independent Duplicate Detection step**

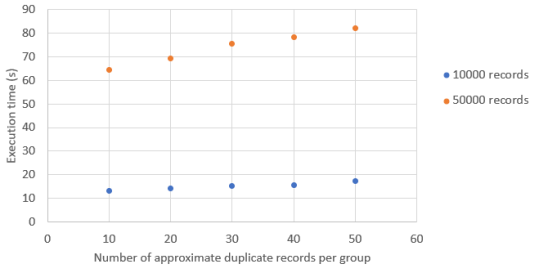


**Figure 9: Execution time for increasing number of approximate duplicates per record for the Domain-Independent Duplicate Detection step**

For the Approximate Duplicate Detection, we used always the same step configuration, only varying the partitioning attributes. We collected execution times for the cartesian product in the four smaller datasets and performed blocking with one attribute for all the datasets. The Approximate Duplicate Detection step, when performing the cartesian product presents an exponential growth with the number of records, as it would be expected, as shown in Figure 10. When performing the cartesian product, the step reveals scalability problems, since it is not able to compute the approximate duplicate records for datasets of 5000 records or more. When performing blocking with an attribute, the execution time greatly decreases and presents a less accentuated growth with the increase of the number of records, as illustrated in Figure 11. When increasing the number of records per group, the growth is close to linear, but presents some variations. These values are presented in Figure 12.

*4.2.1 Discussion.* Concluding, the performance tests for the Domain-Independent Duplicate Detection provided satisfactory results, presenting a linear growth with the number of records, opposed to the exponential growth of performing a cartesian product. The increase of the number of approximate duplicate records per group does not have a big influence on the performance of the step. The Approximate Duplicate Detection step performs well when using

[5]http://users.cecs.anu.edu.au/ Peter.Christen/Febrl/febrl-0.3/febrldoc-0.3/manual.html

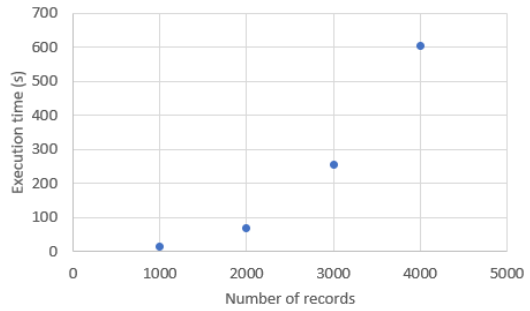


**Figure 10: Execution times for the Approximate Duplicate Detection step performing the cartesian product**

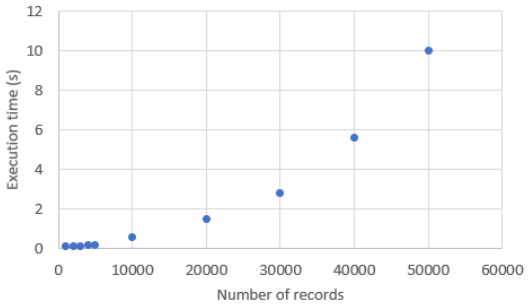


**Figure 11: Execution times for the Approximate Duplicate Detection step partitioning the data by one attribute**

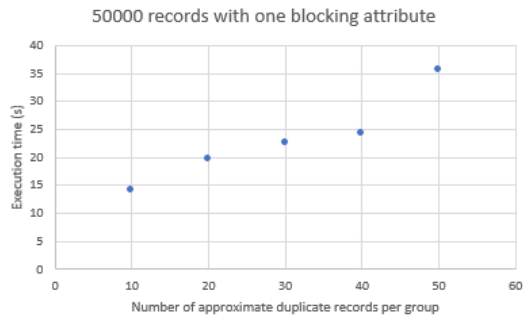


**Figure 12: Execution time for increasing number of approximate duplicates per record for the Approximate Duplicate Detection step with 50000 records**

the partitioning strategy but does not perform well when the cartesian product is made. Although the obtained times for the cartesian product performed in our step are smaller than the ones obtained with another step in PDI, the step can only compute the results for datasets up to 5000 records, which presents a scalability problem when no blocking attribute is provided. Still, our objective to provide more efficient alternatives to the cartesian product was achieved.

## 4.3 Usability

To assess the usability of our solution, we performed user tests with 20 PDI users. These users presented different degrees of familiarity with both PDI and the approximate duplicate detection domain. The users were shown a video introducing the approximate duplicate detection concept and demonstrating the main functionality of our solution. Then, the users were asked to perform 5 tasks. For each of the tasks, we collect the number of errors each user made while configuring the steps, the execution time of the task and whether the task was completed successfully. The distribution of the users' familiarity with PDI and the approximate duplicate detection domain is illustrated in Figure 13. A summary of the results is presented in Table 4.

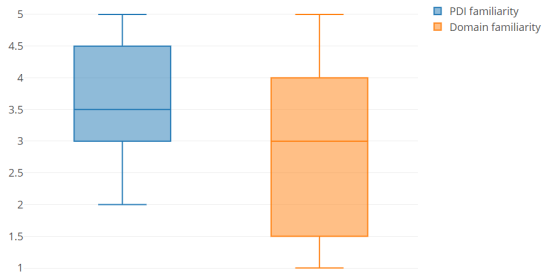


**Figure 13: Distribution of the users' familiarity with PDI and the approximate duplicate detection domain**

| | | Average | Standard Deviation |
|---|---|---|---|
| Task 1 | Success | 75% | |
| | Errors | 10% | |
| | Time | 3m 40s | 2m 17s |
| Task 2 | Success | 85% | |
| | Errors | 0.45 | |
| | Time | 3m 29s | 1m 16s |
| Task 3 | Success | 60% | |
| | Errors | 0.3 | |
| | Time | 4m 20s | 1m 58s |
| Task 4 | Success | 90% | |
| | Errors | 0 | |
| | Time | 5m 17s | 2m 48s |
| Task 5 | Success | 95% | |
| | Errors | 0.15 | |
| | Time | 6m 44s | 2m 48s |

**Table 4: Average values for the task results**

The users were also given a questionnaire to evaluate their satisfaction with the work developed. For the Domain-Independent Duplicate Detection step, the users were overall happy, assigning, on average, 4.5 out of 5 to the step being easy to use, 4.3 out of 5 to the parameters being easy to configure and 4.6 out of 5 to the results being easy to understand. For the Approximate Duplicate Detection step, the users assigned, on average, 4.5 out of 5 to the step being easy to use, 4.0 out of 5 to the parameters being easy to configure and 4.4 out of 5 to the results being easy to understand.

For the Table visualization, the feedback was overall positive, although some users did not find the need to resort to the Table, using alternatives within PDI. We did not include the feature that was proposed by the users, but recognize that it is relevant to make the Table visualization more useful. Even so, the users considered it to be useful, assigning an average of 4.7 out of 5 to the corresponding question in the questionnaire.

*4.3.1 Discussion.* By the analysis performed to the tests results, we can infer that there is no correlation between performing the tasks correctly and the familiarity with either PDI or the domain. Concluding, we can say that we have achieved our goal to create steps that would allow PDI users to detect approximate duplicates in an easy way, since the results of the tests were overall positive, where users that were both familiar and not familiar with PDI were able to perform most of the proposed tasks without major difficulties.

## 5 CONCLUSIONS

In this work we proposed a solution to face the lack of data profiling capabilities of PDI, focusing on approximate duplicate detection. We detailed how we added new approximate duplicate detection functionality to PDI by creating two PDI steps, as well as how we modified an already existing visualization to help in the interpretation of results. We also presented the evaluation results for our solution, in terms of correctness, performance and usability.

With this work, we can conclude that our objective was achieved, considering the implemented solution provides more efficient alternatives to computing the cartesian product of the data, still yielding acceptable results. Users were able to perform most of the proposed tasks successfully, so we achieved our goal to offer a solution that was easy to use for PDI users. Our solution enables users that are not very familiar with PDI to detect approximate duplicates, since the previously existing possibilities to detect approximate duplicates consist in building transformations with several PDI steps. With the Domain-Independent Duplicate Detection step, we provide the users with a solution that can be executed without the need to provide any configurations regarding the dataset being processed. With the Approximate Duplicate Detection step, a user that has some knowledge regarding the dataset being processed can configure the step to produce more consistent results. Both the steps will be made available to the Pentaho community in the Pentaho Marketplace[6].

### 5.1 Limitations and future work

After evaluating the steps and the visualization, and receiving feedback, we identified some aspects that could have been useful to include and that may be used to improve our solution in the future.

Regarding the visualization, one aspect that was identified was the inability to order the data through multiple columns simultaneously. This would allow the users to order the data within a group if they were using the *Group By* visual role. For example, if a user assigned the attribute *name* to the *Group By* visual role, the records would be ordered by that attribute and, if a secondary ordering was possible, the user could then choose, for example, the *Group* attribute that is outputted by our steps to order and be able to see the approximate duplicate groups within each value of the *name* attribute.

For the Approximate Duplicate Detection step, while doing the performance evaluation, we encountered a problem for larger datasets. While the step performs well when at least one attribute is used for blocking, if a cartesian product is performed, the step is not able to finish processing the data and computing the approximate duplicate groups. This is a matter that should be corrected in the future, to allow the user to use the step with large datasets, even if they do not choose an attribute to partition the data. A possible solution for this problem is to implement another partitioning strategy that will restrict the number of comparisons.

Finally, in the future we can implement one of the users' suggestions regarding the steps output. This change would provide the option to divide the step output in two steams, where one stream would, for example, have the records that were grouped with others, and the other stream would have the records that were not grouped with any other record.

[6] https://marketplace.hitachivantara.com/pentaho/

## REFERENCES


[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling Relational Data: A Survey. *The VLDB Journal* 24, 4 (2015), 557–581.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[3] AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of data integration.* Elsevier.

[4] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. 2007. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering* 19, 1 (2007), 1–16.

[5] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2010. Introduction to information retrieval. *Natural Language Engineering* 16, 1 (2010), 100–103.

[6] Alvaro Monge and Charles Elkan. 1997. An efficient domain-independent algorithm for detecting approximately duplicate database records. (1997).

[7] Alvaro E Monge. 2000. An adaptive and efficient algorithm for detecting approximately duplicate database records. *On-line document, URL: http://citeseer. nj. nec. com/monge00adaptive. html* (2000).