Detecting Duplicate Objects in XML Documents

Melanie Weis
Humboldt-Universität zu Berlin
Berlin, Germany
mweis@informatik.hu-berlin.de

Felix Naumann
Humboldt-Universität zu Berlin
Berlin, Germany
naumann@informatik.hu-berlin.de

ABSTRACT

The problem of detecting duplicate entities that describe the same real-world object is an important data cleansing task, necessary to improve data quality. For data stored in a flat relation, numerous solutions to this problem exist. As XML becomes increasingly popular for data representation, algorithms to detect duplicates in nested XML documents are required.

In this paper, we present a domain-independent algorithm that effectively identifies duplicates in an XML document. The solution adopts a top-down traversal of the XML tree structure to identify duplicate elements on each level. Pairs of duplicate elements are detected using a thresholded similarity function, and are then clustered by computing the transitive closure. To minimize the number of pairwise element comparisons, an appropriate filter function is used. The similarity measure involves string similarity for pairs of strings, which is measured using their edit distance. To increase efficiency, we avoid the computation of edit distance for pairs of strings using three filtering methods subsequently. First experiments show that our approach detects XML duplicates accurately and efficiently.

Keywords

Duplicate Detection, XML, Similarity, Data Cleansing

1. INTRODUCTION

Several problems arise in the context of data integration, where data from distributed and heterogeneous data sources is combined. One of these problems is the possibly inconsistent representation of the same real-world object in the different data sources. When combining data from heterogeneous sources, the ideal result is a unique, complete, and correct representation for every object. Such data quality can only be achieved through *data cleansing*, where the most important task is to ensure that an object only has one representation in the result. This requires the identification of duplicate objects, and is referred to as *object identification* or *duplicate detection.*

The problem has been addressed extensively for relational data stored in tables. However, relational data only represents a small portion of today's data. Indeed, XML is increasingly popular as data representation, especially for data published on the World Wide Web and data exchanged between organizations. Therefore, we need to develop methods to detect duplicate objects in nested XML data. Due to the hierarchical and semi-structured nature of XML data, we have to face the following two issues, namely *element scope* and *structural diversity.*

As XML elements, which we consider as objects, can be nested under other XML elements, their scope is restricted according to their ancestors, and has to be taken into account. For instance, consider the XML elements in Figure 1.

```
<country>
       France
       <city>Paris</city>
</country>
<country>
       USA
       <city>Paris</city>
</country>
```

Figure 1: Element Scope

Both cities are called Paris, both their content and structure match, however, they are not the same because one city is in France, the other one in the USA, which are obviously different.

Another problem is the a priori undefined structure of an XML element within a document. In relations, tuples in a single relation all have the same, clearly defined structure. However, XML elements describing the same type of object may have different structures. As a consequence, two different structures do not necessarily mean that the elements represent different real-world objects: the same data may just be organized differently or may be incomplete, which the Example in Figure 2 illustrates.

The two country elements represent the same object, namely the USA, but their structures differ significantly. To identify them as duplicates, we have to loosen the condition of exact structural match.

In this paper, we present a novel, domain-independent approach that overcomes these problems and efficiently detects duplicates in an XML document. More specifically, our ap-

```
<country>
        <name>United States of America</name>
        <cities>New York, Los Angeles, Chicago</cities>
        <lakes>
                <name>Lake Michigan</name>
        </lakes>
</country>
<country>
        United States
        <city>New York</city>
        <city>Los Angels</city>
        <lakes>
                <lake>Lake Michigan</lake>
        </lakes>
</country>
```

Figure 2: Structural Diversity

proach detects duplicate objects in a single XML document, assuming the structure of elements with equal name may differ. Without further domain-specific information, we are able to detect duplicate objects with typographical errors (data differs only slightly), equivalence errors (data differs significantly but has same meaning), and missing data.

The outline of this paper is as follows. In Section 2, we present related work. In Section 3, we discuss the main concepts and definitions on which our approach is based. Section 4 provides a detailed description of our approach. In Section 5, we present experimental results.

2. RELATED WORK

Data cleansing is an important task in improving data quality, and receives increasing attention in the database community [22, 8, 21, 26]. For the specific problem of deduplication, a lot of work is available for relational data. Recent work focusing on performance aspects assuming user input deduplication functions includes [10, 17].

In some domains, detecting duplicates received particular attention, and domain-dependent solutions were developed. Examples include Census datasets [25], medical data [12], genealogical data [20], and bibliographic data [11]. Domain-independent solutions include [27, 16, 14, 18]. Some approaches rely on learning algorithms to improve similarity measuring [23, 6, 7].

To the best of our knowledge, the closest approach for detecting duplicates in hierarchical data is DELPHI, from which our work is inspired [3]. DELPHI identifies duplicates in the hierarchically organized tables of a data warehouse, for which the structure is clearly defined. This is a big difference to XML, where the structure is a priori not known. They focus on a single hierarchy, not considering the cases where a table may have several children tables. Our approach and DELPHI further differ in the similarity measure used to identify duplicates. In DELPHI, they chose an unsymmetrical measure (i.e. A is duplicate of B not necessarily means that B is duplicate of A) that measures the containment of an element within another. As a consequence, the difference of the two elements is not reflected in the result. Our similarity measure, as defined in Section 3.2 is both symmetrical and takes into account differences in the compared elements.

Efficient string matching [13, 19, 5] is crucial when considering large data sets. We also consider this issue in our approach by employing three different edit distance filters.

3. CONCEPTS AND DEFINITIONS

3.1 Definition of Duplicates

Before we start searching for duplicate objects in an XML document, it is essential to define the exact meaning of "duplicate XML object". In the relational model, data is stored in flat tables where an object is simply defined as a single row of such a relational table. All objects in the table have the same structure and thus only vary in content. In the world of XML, there are not necessarily uniform and clearly defined structures like tables. Two objects thus have to be compared according to both structure and data. XML data is organized hierarchically, and the structure, even when constrained by an XML Schema, allows a certain degree of freedom. How objects that apply for being duplicates are defined under these circumstances is the subject of this section.

The two main aspects we consider to detect duplicate XML objects are structure and data.

- *Structure*: Elements, which are the containers for data, are distinguished by name. In our approach, we assume that when two elements have different names, they also have different semantics, i.e., the data they contain cannot represent the same real-world entity.

  Two elements with equal element name may be nested under other elements, their ancestors. If two elements have different ancestors, we assume that they cannot be duplicates. We thus solve the element scope problem.

  The descendents of two elements that satisfy the previous conditions, may have different structures, although they represent the same real-word entity. To account for this possibility, we only require descendent structures to be similar, but not necessarily equal.

- *Data*: In XML, there are two choices storing data. It can either be specified as value of an attribute, or as text node of an element. When comparing two XML objects, the both should be considered.

  Data is organized hierarchically. To identify duplicates even if the data significantly differs at one level due to an equivalence error, a good indicator that they may actually represent the same real-world entity is given by the data of their children. Indeed, if the data of children nested under an element is similar to the data of children of the other element, this is strong evidence that the elements are duplicates. We say that the children elements common to two elements *co-occur* in both elements. Common children elements have equal element name and similar data.

  The definition of an element's data is application dependent. E.g, it may be sufficient to consider the text node and attribute values of an element, however, XML elements with complex content (i.e., only sub-elements) will have no data. In this case, it is preferable to consider the text nodes of children elements as the element's data. To support all types of content models (simple, complex, mixed), we defined the data of an element to consist of (i) the data of its text node, (ii) the values of its attributes, (iii) the data of its children text nodes.

From these observations, we formally define duplicate objects in XML as follows.

DEFINITION 1. *Two XML elements e and e' are candidate duplicates if the following conditions are satisfied:*

- *the parent elements of e and e' are equal or similar*
- *e and e' have the same name*
- *the data of e and e' is similar*
- *the children sets of e and e' have similar structure and contain similar data*

In each case, similarity is defined by some function that is described in the next section.

3.2 Similarity Measure

Our technique for detecting similar objects uses a thresholded similarity measure. That is, two objects are considered similar if the similarity measure yields a result above a given threshold. Our similarity measure is guided by the following intuition. Let O and O' be two multisets of objects in a universe U. The larger the intersection between O and O', the more similar they are. On the other hand, the larger their multiset difference compared to their intersection, the more different they are.

We further adopt the notion that objects may have different relevance in distinguishing O and O', which is quantified by their *inverse document frequency* (IDF). The use of the IDF to quantify the notion of importance has been successfully used in information retrieval literature [4]. The IDF is defined as follows. Let $f_S(o)$ denote the frequency of an object $o \in O$. Then, $IDF_S(o) = log(\frac{|U|}{f_s(o)})$. As in [3], we further define the IDF of a set $S \subseteq O$, to be

$$IDF_S(S) := \sum_{o \in S} IDF(o).$$

We introduce a similarity measure comparing two string sets, denoted S and S'.

$$sim(S, S') := \frac{IDF(S \cap S')}{IDF((S \cup S') \setminus (S \cap S')).} \quad (1)$$

According to Definition 1, the calculation of two XML elements' similarity requires the determination of (i) the similarity between data contained in XML elements, called *element data* and (ii) the similarity of their *children data* in order to measure co-occurrence. Of course, two children elements can only co-occur if they have same name, that is, we have to consider their structure as well.

Both element and child data are considered as strings. Element data is further divided into a set of tokens. Two tokens or strings s and s' are considered similar if their edit distance $d_{edit}(s, s')$ divided by the maximum length of s and s' is below a given threshold t_{edit}. The edit distance is a common measure for string similarity and is defined as the minimum number of insert, delete, and replace operations necessary to transform s into s'. We divide $d_{edit}(s, s')$ by the maximum length of s and s' because more errors should be allowed in longer strings (intuitively, the longer a word, the more errors such as typographical errors may occur).

Let E denote a set of elements that need to be compared. For an element $e \in E$, $TS(e)$ denotes the token set comprising the data in e, and $CS(e)$ denotes the set of strings composing e's children data. The similarity of two elements e and e' is then defined as

$$s(e, e') = sim(TS(e) \cup CS(e), TS(e') \cup CS(e')) \quad (2)$$

. We extend the classical definition of set intersection to include not only equal strings, but also tokens and children considered similar according to their edit distance. The efficient determination of this similarity employs the concepts described next.

3.3 Identifying Similar Data

The similarity of two elements is based on the similarity of their data as well as their children data. In both cases, the data is considered as strings, so we need to determine pairwise string similarity, measured using edit distance. Computing the edit distance of two strings is an expensive operation. By applying the following general edit distance filters for pairs of tokens and pairs of children data, the number of edit distance computations can be substantially reduced. Edit distance filtering is applied once the graph is initialized.

3.3.1 *Length Distance Filter*

When comparing two strings s and s' of length $l(s)$ and $l(s')$, it is true that the following inequality holds:

$$|l(s) - l(s')| \leq d_{edit}(s, s'). \quad (3)$$

This property was already used in [9]. In our approach, we first group strings by length, and we then prune out complete groups of string pairs that do not qualify to be similar: let L be the group of all strings of length l, and L' the group of all strings of length l'. As a reminder, two strings s and s' are duplicates if $d_{edit}(s, s') < t_{edit}$. Now, if $|l - l'| \geq t_{edit}$, there exists no string $s \in L$ that is a duplicate for a string $s' \in L'$, because Equation 3 holds for every s and s'. Therefore, we can prune out pairs of string groups, each time saving saving $|L| * |L'|$ edit distance operations by computing a single difference.

3.3.2 *Filtering using Triangle Inequation*

A second method for saving expensive edit distance calculations makes use of the triangle property that holds for edit distance. Let x, y , and z be three strings. It can be shown that following inequality holds:

$$|d_{edit}(x, y) - d_{edit}(y, z)| \leq d_{edit}(x, z) \leq d_{edit}(x, y) + d_{edit}(y, z) \quad (4)$$

We can use this inequality to calculate a range $[min, max]$ for $d_{edit}(x, z)$ by computing a simple substraction and addition. This is cheaper than calculating the edit distance. Then, there exist two filter methods:

1. $max < t_{edit} \Rightarrow x$ and z are similar
2. $min \geq t_{edit} \Rightarrow x$ and z are not similar

3.3.3 *Bag Distance Filter*

For the remaining strings, we use the bag distance between two strings, which was introduced in [5] as a lower bound for the edit distance of those strings. Given a string x over an alphabet A, let $X = ms(x)$ denote the multiset of symbols in x. For instance, $ms("peer") = e, e, p, r$. Let the bag distance be defined as follows:

$$d_{bag}(x, y) = max(|X - Y|, |Y - X|) \quad (5)$$

where the difference has bag semantics (e.g., $\{\{a, a, a, b\}\} - \{\{a, a, b, c, c\}\} = \{\{a\}\}$), and $|.|$ counts the number of elements in a multiset (e.g., $|\{a, a\}| = 2$). In practice, $d_{bag}(x, y)$ first drops common elements, then takes the maximum considering the number of residual elements. It can be easily shown that $d_{bag}(x, y) \leq d_{edit}(x, y)$, so it is a potential filter function for edit distance. Its use is justified as its computation in $O(|X| + |Y|)$ is substantially cheaper than the calculation of the edit distance performed in $O(|X| * |Y|)$.

Once similar tokens and children have been identified, we can apply the similarity measure (2) to identify pairs of duplicate XML objects.

3.4 Detecting Pairs of Duplicate Objects

As mentioned earlier, we use a thresholded approach to detect pairs of duplicate objects. Formally, let t_{dup} be a threshold value, and $isDup(e, e')$ be a function returning a boolean value such that

$$isDup(e, e') = \begin{cases} TRUE & \text{if } s(e, e') > t_{dup} \\ FALSE & \text{otherwise} \end{cases} \tag{6}$$

If $isDup(e, e')$ yields a positive result, e and e' are considered duplicate elements.

Consider again the two elements of Figure 2. Set $t_{edit} = 0.15$ and $t_{dup} = 1.0$. The intersection of both elements is {`"United"`, `"States"`, `"New"`, `"York"`, `"Los"`, `"Angeles"`, `"Lake Michigan"`}. `"Angeles"` is part of the intersection because d_{edit}(`"Angeles"`, `"Angels"`) $< t_{edit}$. The difference between the two elements consists of {`"of"`, `"America"`, `"Chicago"`}. Assuming all tokens and children data have equal IDF , we obtain a similarity $s = 7/3 > t_{dup}$, so both elements are considered duplicates.

As $s(e, e')$ is applied to pairs of elements, the number of comparisons explodes for a large number of elements. Similar to our approach for reducing the number of edit distance computations, we developed a filter function for s, described next.

3.5 Object filter

We apply the following filter function to reduce the number of expensive pairwise object comparisons. The filter function represents an upper bound to our similarity measure s. With e being an XML element, $S(e) = TS(e) \cup CS(e)$ being the set of strings composing e's data, and G being the set of data strings of all elements, the filter function f is defined as:

$$f(e) = \frac{IDF(S(e) \cap (G - \{S(e)\}))}{IDF(S(e) \setminus (S(e) \cap (G - \{S(e)\})))} \tag{7}$$

Informally, $f(e)$ considers all data that e shares with any other element, relative to data unique to e. Thus, if e shares very few data with any other element in G, $f(e)$ yields a small result. As a consequence, it is likely that e is no duplicate of any other element, because it is too isolated. Formally, it can be easily shown that

$$s(e, e') \leq f(e) \tag{8}$$

for any $e' \in G$. As a reminder, two elements are considered duplicates if $s(e, e') > t_{dup}$. If $f(e) \leq t_{dup}$, it follows from (8) that $s(e, e') \leq f(e) \leq t_{dup}$ for any $e' \in G$, so we can conclude that e has no duplicates without calculating any similarity for e. The cost of computing $f(e)$ is comparable to the cost of calculating s. However, f only needs to be calculated once for every element, whereas s has to be computed for every pair of elements. Therefore, $f(e)$ is a suitable filter for reducing the number of pairwise element comparisons.

So far, we have seen which measures are required in order to determine pairs of duplicate objects. We use these measures to reach our broader goal of efficiently identifying all duplicate elements at different levels in an XML document.

4. GENERAL APPROACH

The general approach for detecting duplicate XML objects compares elements with the same name in a top-down traversal of the XML structure. A top-down traversal is necessary to ensure that duplicates of parent elements are detected prior to comparing their children. In our approach, the detection of duplicate elements of same name is divided in six major steps, which are listed below. In the remainder of this section, we will describe these steps in more detail.

1. *Object Extraction*: creates an XML document that only contains the relevant information for comparisons.

2. *Graph Generation*: creates an internal graph representation of the considered XML objects.

3. *Similar Data Detection*: detects similar tokens and similar children data efficiently. To this end, the edit distance filters described in Section 3.3 are applied.

4. *Object Filter*: applies the object filter described in Section 3.5, to minimize number of pairwise object comparisons.

5. *Pairwise Object Comparison*: performs pairwise object comparisons to detect duplicate objects. An edge is added between element vertices that represent duplicate elements. The set of edges containing all these edges is denoted as E_d.

6. *Duplicate Clustering*: computes the transitive closure over the graph $G(V, E_d)$ in order to obtain clusters of duplicate elements.

The input for the first iteration of this processing pipeline is the XML document in which duplicates should be detected. In subsequent iterations, i.e. during the top-down traversal, these processing steps are repeated with a cluster as additional input to the Object Extraction phase. This way, we ensure that only children elements of the duplicate elements in one cluster are considered and all the others are out of scope.

Let us now look at the different steps in more detail.

4.1 XML Object Extraction

The XML document in which we want to detect duplicate objects contains elements of various names that are nested at different levels. According to our definition of duplicates, it is sufficient to search for duplicates in a smaller domain, where elements have the same name and ancestors. Using XQuery, we extract all objects of the desired domain and write them to a new XML document, which we refer to as *object document*. The object document has a predefined schema that correlates elements to their token and children sets. The benefit of this preprocessing is that the document we have to parse in order to generate the graph has a clearly defined structure, thus making the graph generation a straightforward task. We use XQuery instead of writing our own parser, because XQuery simplifies the task of keeping track of current elements' XPaths and groups of parent elements.

The general XQuery used to extract objects from the initial XML document is shown in Figure 3. Adapting this query to differently structured documents is straightforward, as it only requires changing line 3, containing the name of the input document and the XPath of elements to be compared.

```
(1)  <result>
(2)  {
(3)   for $element in doc(inputDoc)/elementXPath
(4)   return
(5)   <element>
(6)   {
(7)     (
(8)      $element/text(),
(9)      for $att in $element/@* return data($att),
(10)     for $child in $element/element()
(11)       let $childatts := $child/@*
(12)       return
(13)       (
(14)         $child/text(),
(15)         for $childatt in $childatts return data($childatt),
(16)         <children>
(17)          {
(18)            for $child2 in $child/element()
(19)              let $childatts2 := $child2/@*
(20)              return
(21)              element {local-name($child2)}
(22)              {
(23)                 $child2/text(),
(24)                 for $att2 in $child2/@* return data($att2),
(25)                 for $child3 in $child2/element()
(26)                   let $childatts3 := $child3/@*
(27)                   return ($child3/text(),
(28)                 for $childatt4 in $childatts3
(29)                    return data($childatt4))
(30)               }
(31)          }
(32)         </children>
(33)       )
(34)     )
(35)  }
(36)  </element>
(37) }
(38) </result>
```

Figure 3: XQuery Generating Object Document

We briefly explain the general-purpose XQuery of Figure 3. The set of objects we want to compare is a set of XML elements from the input XML document *inputdoc*, whose XPath is *elementXPath* (l.3). For every element e, an element named `element` is created, the text node consisting of the concatenation of e's attribute values (l.9), text node (l.8), and attribute and text nodes of e's direct children (l.10-15), thus comlying to the definition of element data provided in Section 3.1. A `children` element is created under every `element`. It encompasses the structure one level below e by creating elements of same name (l.21), and the data of these children (l.23-29). In short, we flatten the structure of considered XML objects from the input XML document from four levels to two. The data of levels 1 and 2 and levels 3 and 4 in the original XML document is condensed on level 1 and 2, respectively. By losing some structure and keeping the data, we loosen the constraint in structural similarity. It follows that we can detect duplicates based on similar data, although they may have significantly different structure (as in Figure 2). The object document for this sample XML document, when considering `country` elements, is shown in Figure 4.

```
<element>
       United States of America
       New York, Los Angeles, Chicago
       <children>
              <lakes>Lake Michigan</lakes>
       </children>
</element>
<element>
       United States
       New York
       Los Angels
       <children>
              <lakes>Lake Michigan</lakes>
       </children>
</element>
```

Figure 4: Sample Object Document

By parsing this document, which is potentially smaller than the original XML document, we create an in-memory representation of the XML objects. Future work will address the memory problem arising for large XML documents.

4.2 Graph Model

The internal representation used in our approach is a graph structure. Let $G(V, E)$ be a graph with V being the set of vertices and E the set of edges. In V, we distinguish vertices describing an element e, vertices representing tokens of e and vertices that represent children of e. These are referred to as *element vertices*, *token vertices* and *children vertices*, respectively. The correspondence of a token or a child to an element is represented by an edge between the corresponding token or child vertex and element vertex.

The graph is set up by parsing the object XML document obtained in the previous step, using a SAX parser. For every encountered `element` tag, an element vertex v_e is created. We keep track of data following the element tag until the `children` tag is encountered. This data is then divided into tokens (separators are whitespaces). For each token, a token vertex is initialized and an edge between v_e and the token vertex is added. While no closing `children` tag is encountered, a child vertex is initialized for every XML element, and an edge between v_e and the child vertex is added.

Considering the object document in Figure 4, we obtain the graph depicted in Figure 5.

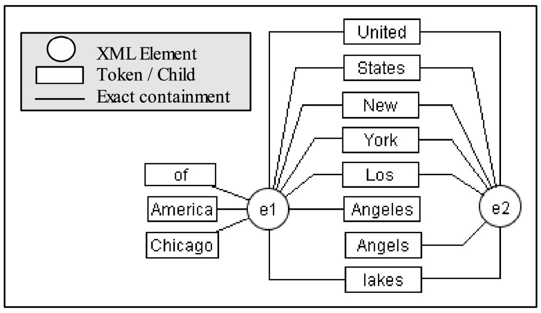

Figure 5: Graph Representation of Figure 4

4.3 Similar Token and Children Detection

In order to determine the similarity of two objects, it is necessary to determine the similarity of their data and their children data. To achieve this goal, we detect pairs of tokens and children data that are similar according to their edit distance. Clearly, it is impractical to compute the edit distance for all pairs of tokens and children. To minimize the number of comparisons using edit distance, we employ the filters for edit distance introduced in Section 3.3 as follows. Note that the discussion focuses on token comparisons, but the approach is applicable to the comparison of children data, as well.

The first filter applied to the data is the length filter. This is reasonable as it is very easy to compute and prunes out pairs of sets of tokens instead of just pairs of tokens. Furthermore, the selectivity of this filter is high, as will become clear from Experiments in Section 5. For every pair of token sets T and T' not pruned out by the length filter, we reduce the number of necessary edit distance computations by combining the triangle property filter and the bag distance filter. How they are combined is best illustrated using an example.

Let us consider the following pair of token sets not pruned out by length distance filter :

```
T = {Oakland, Houston, Gillroy, Oaklamd}
T' ={Gilroy, Austin, Tuscon}
```

We create a matrix $|T| \times (|T| + |T'|)$ and calculate all edit distances in the first row (see Figure 6). For the remaining cells, we first apply the triangle inequality filter using two exact edit distances from row 1. If the triangle inequality filter fails, the bag distance filter is applied next. If it also fails, the edit distance is finally calculated. In Figure 6, we see that the token pairs {("Oaklamd", "Houston"), ("Oaklamd", "Austin"), ("Oaklamd", "Houston"), ("Oaklamd", "Gillroy"), ("Oaklamd", "Gilroy"), ("Oaklamd", "Tuscon")} are filtered using the triangle inequality filter. The bag distance filter prunes out all but one of the remaining token pairs. For the non-filtered ("Gilroy", "Gillroy") pair, the edit distance is calculated. Through filtering we saved 11 out of 18 edit distance calculations in the current example. It is guaranteed that these computations are applied only once for every existing pair of tokens in the complete token set, but the details are not further discussed.

	Oakland	**Houston**	**Gillroy**	**Oaklamd**	**Gilroy**	**Austin**	**Tuscon**
Oakland	■	d_{edit} = 7	d_{edit} = 6	d_{edit} = 1	d_{edit} = 6	d_{edit} = 6	d_{edit} = 6
Houston	■	■	min = 1 d_{bag} = 6	min = 6	min = 1 d_{bag} = 6	min = 1 d_{bag} = 3	min = 1 d_{bag} = 3
Gillroy	■	■	■	min = 5	min = 0 d_{bag} = 1	min = 0 d_{bag} = 6	min = 0 d_{bag} = 6
Oaklamd	■	■	■	■	min = 5	min = 5	min = 5

░ Token pairs pruned out using triangle inequality ▒ Token pairs pruned out using bag distance filter □ Token pairs pruned out using bag distance filter

Figure 6: Edit Distance Filtering

If two tokens represented by token vertices v_t and v_t' are duplicates and are part of the data of two elements represented by element vertices v_e and v_e', we add two directed edges (v_t, v_e') and (v_t', v_e) to G. The direction is included so that only one of two similar tokens is included in the intersection of both elements when calculating $s(v_e, v_e')$. Figure 7 illustrates the graph representation of the XML elements of Figure 2. The similarity of tokens "Angeles" and "Angels" is translated by the two directed edges ("Angeles", e2) and ("Angels", e1).

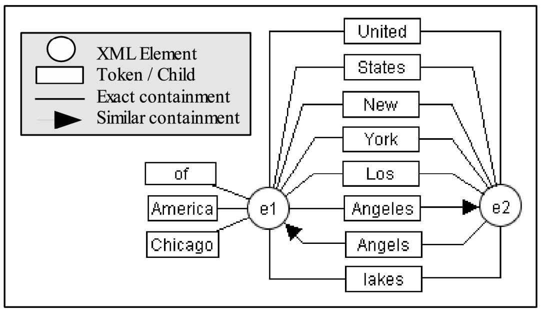

Figure 7: Graph Representation of Figure 2 after Similar Data Detection

Principally, we could now calculate the similarity of every element pair, however, this is impracticable for a large number of elements. Therefore, we first apply our object filter (7) to prune out elements that have no duplicates.

4.4 Object Filtering

The filter function f is applied to every element vertex. Let v_e be the current element vertex representing the XML element e. The intersection and set difference are computed efficiently by simply looking up which of v_e's neighbors have a degree larger than 1, and those that have a degree equal to 1, respectively. When v_e is filtered, it is removed from G together with its edges.

Removing an element vertex v_e from G reduces $f(n)$ for any element n that intersected with e prior to v_e's removal. Therefore, it is possible that $f(n) \leq t_{dup}$ after v_e's removal, although $f(n) > t_{dup}$ before. Through removing element vertices from G, the number of potentially filtered element vertices increases.

For example, consider the graph shown in Figure 8, where v and v' respectively represent elements e and n. Assuming all tokens have equal IDF, we initially obtain $f(e) = 1$ and $f(n) = 3/2$. For $t_{dup} = 1$, v is filtered and removed from G, which results in the bottom graph in Figure 8. By removing v, $f(n)$ drops to $1/4$, which is below t_{dup}. So we see that v' can be filtered after the removal of v, which was not possible before v was filtered.

As the goal is to maximize the number of filtered elements, we recompute f for every element vertex that intersected with v_e prior to its removal. The implementation involves a filter queue, to which we initially add all element vertices. When an element is filtered, those neighbors not already present in the queue but still in G, i.e., those that were processed and not filtered, are added to the queue so that f is recomputed.

All elements not filtered during this phase possibly have duplicates. The detection of duplicates is the goal of the next phase.

4.5 Duplicate Detection

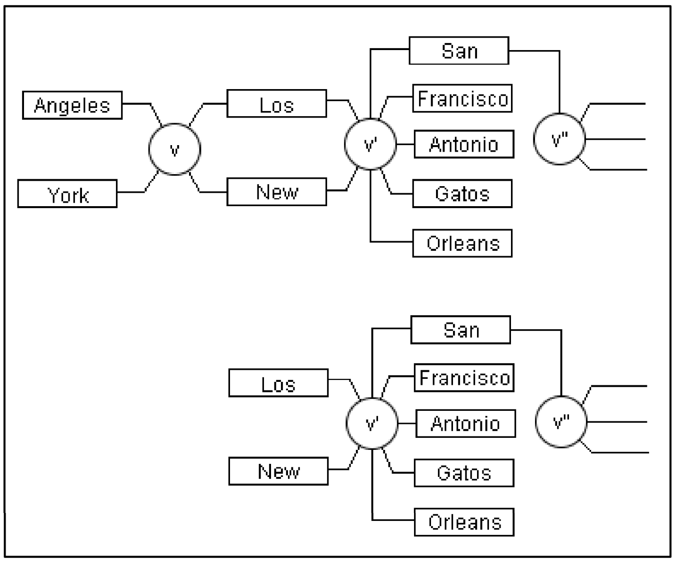

Figure 8: Graph before filtering of v_e (top) and after removing v_e (bottom)

Let $v_e, v'_e \in V$ be the element vertices in graph $G(V, E)$ representing XML elements e and e', respectively. We assume they have not been pruned out during the object filtering phase. Their token and children sets are simply their neighboring token and children vertices. The similarity of e and e' is measured by calculating $s(e, e')$. The naive approach to do this would require $|V_e| * (|V_e| - 1)/2$ comparisons, where V_e is the set of element vertices in G that have not been filtered. However, the determination of the set N_{ev} of element vertices sharing at least one neighbor with an element vertex v_e, significantly improves performance. Then, s only has to be computed $|N_{ev}|$ times for v. This is by far cheaper than the naive approach, because $|N_{ev}|$ is usually small, compared to $|V_e|$.

When e and e' are detected to be duplicates according to $s(e, e')$, an edge (v_e, v'_e) is added to the graph G. Once all duplicate pairs have been identified, the transitive closure over these edges is computed, and we obtain clusters of duplicate elements. Note that these edges cannot be confused with other edges in the graph because they are the only edges between two element vertices. All elements in a cluster represent the same real-world entity.

4.6 Top-Down Traversal

Once we have detected all duplicate clusters of elements on the same level, we can descend one level in the XML hierarchy to search for duplicates. For each of the determined clusters, the processing pipeline described above is repeated, the selection of relevant XML objects being based on the duplicate clusters of the previous iterations. The relevant XML objects per iteration are those of same name and whose parents are members of the same duplicate cluster. Formally, for every cluster C detected at level i, repeat the pipeline for those elements at level $i + 1$ whose parent is in C.

To evaluate the effectiveness of our approach, we carried out some experiments, described next.

5. EXPERIMENTS

We perform experiments to answer two questions.

1. How effective are the filters described in Section **??** in minimizing the number of pairwise string and object comparisons?
2. How effective is the proposed approach in identifying duplicate XML elements?

In answering the first question, we expect the combination of edit distance filters to substantially reduce the number of necessary pairwise string comparisons. Further, the object filter should prune out the majority of objects not having any duplicate in the considered set of objects. Concerning the second question, we do not expect substantially different results in terms of precision and recall than domain-independent algorithms applied on relational data. However, the problem of identifying duplicates in XML bears additional issues that are well solved according to our precision and recall measurements.

5.1 Data Sets and Setup

We consider clean geographic information extracted from mondialDB.xml [2]. Clean means that there are no duplicate elements in the XML document. We introduce errors in the clean Country data as described next. Since we know the duplicate tuples and their correct counterparts in the erroneous dataset, we can evaluate our duplicate elemination algorithm. In our experiments, we detect duplicates on level 1, where we have 260 Countries in the clean document. Their children sets sum up to 2340 elements on level 2. Because we start from real data, all characteristics of real data — variations in the lengths of strings, numbers of tokens and frequencies of attribute values, co-occurrence patterns, etc. — are preserved.

5.2 Error Introduction

We introduce three types of errors in the data, namely typographical errors, equivalence errors, and missing data. Four parameters are specified for the dirty XML generation. First, we can specify the *percentage of duplicates* (dup) of an object. The three remaining parameters specify (i) the *percentage of typographical errors* (typ) consisting of single character insertions, deletions, and swaps, (ii) the *percentage of deletion errors* (dep) creating missing data, and (iii) the *percentage of equivalence errors* (eqp) within the set of generated duplicates.

The parameter values are set as follows for our experiments.

- dup = 100%
- typ = 20%
- dep = 10%
- eqp = 8%

The dirty XML document thus contains 520 Country elements (260 originals and 260 duplicates). From these 260 duplicates, 20% contain typographical errors, 10% miss data present in the original, and in 8% we introduced equivalence errors. It was was generated using the DirtyXMLGenerator made available by Sven Puhlmann.

5.3 Filter Selectivity

As described earlier, our approach uses two filtering techniques. The first is used to reduce the number of edit distance calculations between pairs of tokens. The second filter is applied on element vertices, and aims at minimizing the number of pairwise element comparisons. In the following, we present the selectivity of both filters on our current document. The numbers are based on filtering at level 1, only.

5.3.1 *Edit Distance Filter*

We consider the token set of all Country elements in the dirty document. It consists of 711 tokens, which would require 252405 edit distance calculations in the naive nested loop approach. Using all three filters presented in Section 3.3, namely the length filter, the bag distance filter, and the triangle filter, the number of edit distance calculations is reduced to 1027. The serialization of the three filters thus yields a selectivity of over 99%. The contributions of the individual filters are described next, and are summarized in Figure 10.

We first apply the length filter, which reaches a selectivity of 83%, as it prunes out 207796 token pairs. The selectivity of the length distance filter highly depends on the variability of token length. To characterize the behavior of the length distance filter, we measure its selectivity by varying the average token length and the length variability (i.e., the number of groups of strings of same length). Figure 9 summarizes the selectivity obtained when varying the average token length (x-axis) for a given number of groups of tokens of same length. We assume that tokens are equally distributed over all groups, with the total number of tokens to compare being 1000.

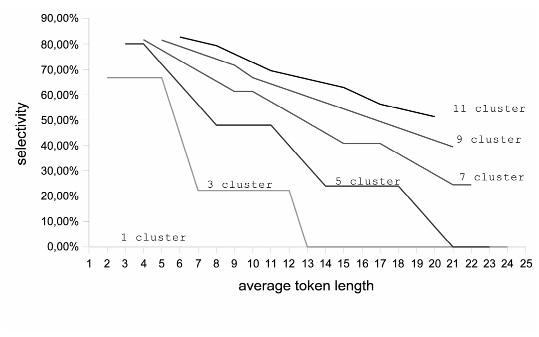

Figure 9: Length Filter Selectivity

We observe that an increase in average token length yields a decrease in the length distance filter's selectivity. This is due to the fact that the threshold is relative to the maximum token length for a pair of tokens (see Section 3.2). We further notice that as the number of groups increases (i.e., when the token lengths are spread more widely), the selectivity increases for constant average string length. The reason for this is that the number of tokens per group, denoted as $|c|$, decreases as the number of groups increases. The number of pairwise token comparisons necessary after length filtering, denoted as n_p, is given by

$$n_p = a/2 * (|c|^2 - |c|) + b(|c|^2)$$

where a is the number of groups and b the number of pairwise groups not pruned out. By increasing a, b potentially increases as well, but the most significant term remains $|c|$. As c decreases with increasing a, n_p decreases.

After applying the length distance filter, 44609 token pairs remain. From these, the triangle filter prunes out 10109 pairs in total (selectivity = 23%). This leaves us with 34500 pairs, from which 33473 are filtered using the bag distance (selectivity = 97%). Future work will include analysis of the effectiveness of the triangle filter despite its relatively low selectivity. Our current belief is that its calculation is so cheap compared to the calculation of bag distance that it is worth calculating it to save bag distance calculations.

Figure 10 summarizes the contributions of the three different filters.

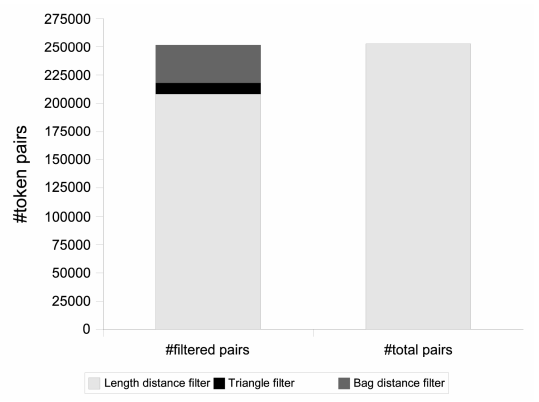

Figure 10: Edit Distance Filter Contributions

We confirmed the results for the edit distance filtering in much larger experiments. As in [14], we used 54,000 movie star names collected from The Internet Movie Database [1]. We obtained results in 1000 seconds, a result comparable to the one in [14]. The advantage of our approach is that we guarantee that all duplicates according to edit distance are found. The mapping of strings to Euclidean space used in [14] only maintains edit distances as well as possible, so that some duplicates are potentially missed.

5.3.2 *Object Filter*

The selectivity of the object filter highly depends on the fraction of duplicate objects in the XML document. Indeed, if every element has a duplicate in the document, the selectivity should be zero, whereas for a clean document, it ideally is 100%.

Using the clean Country document described previously, we again generate dirty XML documents. The parameters are the same as in Section 5.2, except for dup, which is varied from 0% to 100% in increments of 10%. The grey line in Figure 11 describes the number of objects without duplicates in the corresponding document. For high selectivity, the number of objects pruned out by the object filter, described by the black curve, should be close to these values. We see that this is generally the case.

Note that at duplicate percentages 70%, 90% and 100%, the observed number of filtered object exceeds the number of unique objects in the document. That is, the selectivity is higher than 100%. This is due to the fact that f can return a result smaller than t_{dup}, when $s \leq t_{dup}$. In this case, f found a false negative where s would have found one, too.

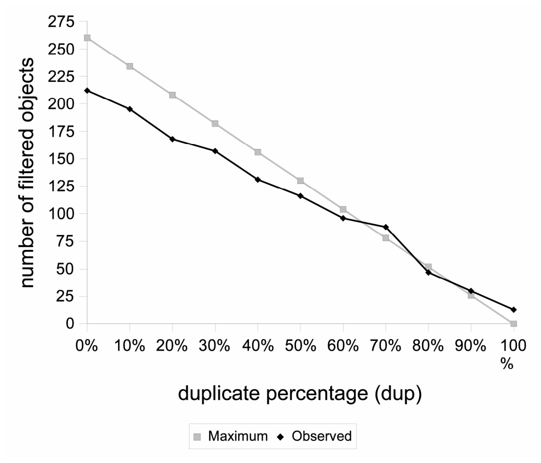

Figure 11: Object Filter Selectivity

5.4 Effectiveness of Similarity Measure

To quantify the effectiveness of our similarity measure s in detecting duplicates, we perform recall / precision analysis [4]. The results are based on duplicate Countries only, that is we do not see error propagation through hierarchies, which is part of future work. We ranked the pairs of duplicates by descending value of s. Figure 12 represents two recall / precision curves. The grey curve shows how recall and precision vary when considering both element and children data. The black curve was obtained by only considering element data, thus leaving out co-occurrence as a measure for similarity. From Figure 12, we can draw two conclusions for the considered scenario.

- the similarity measure is suitable for detecting duplicate XML elements as the precision stays above 80% for recall values up to 80% (with or without the use of cooccurence)
- the use of co-occurrence improves the overall recall / precision result, although the precision is worse than without cooccurence for a recall between between 30% and 70%. This can be explained by the fact that the similarity measure is mislead by the similarity and significance of children data, even when the element data does not match. However, this behavior is a benefit when element data is erroneous due to equivalence errors or missing data. Then, elements can only be similar according to their children data, and are not identified as duplicates without u.

In Figure 12, we summarize the percentages of false positives and false negatives detected with or without considering children data. Again, we see that the use of co-occurrence significantly improves the result, as the number of false positives does not explode. Furthermore, the overall number of false positives and false negatives when considering children data is low, which indicates that s is indeed a suitable measure for detecting duplicate XML elements.

We plan to confirm these promising results by further experiments.

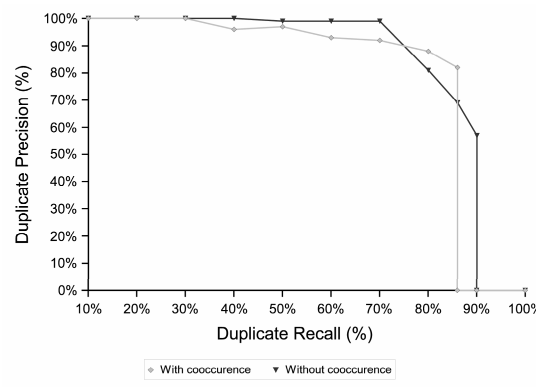

Figure 12: Recall and Precision Diagram

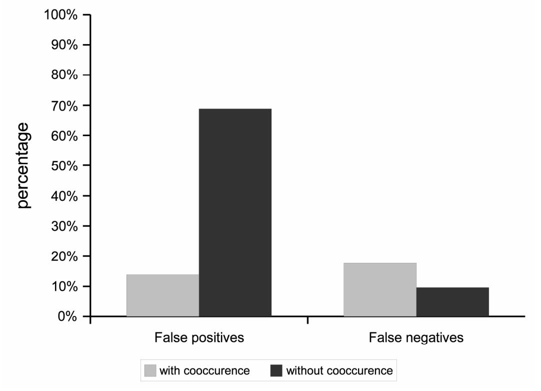

Figure 13: False Positives and False Negatives

6. CONCLUSION AND OUTLOOK

In this paper, we presented an approach to detect duplicate objects in an XML document, which is a very important data cleansing task necessary to improve data quality. Our approach overcomes the problems of element scope and structural diversity within an XML document, and efficiently identifies duplicate elements by adopting a top-down traversal of its hierarchical structure. The results show that our measure for object similarity achieves high precision and high recall in detecting duplicates. The efficiency is improved by applying filtering techniques where we would otherwise have to perform pairwise comparisons. Both the edit distance filters and the object filter are very selective and thus considerably reduce the number of pairwise token and element comparisons.

The approach presented in this paper focuses on a fast and effective duplicate detection using several filtering techniques. In addition to further characterizing filters and the similarity measure, future work will consider issues of memory consumption. We also intend to extend our method to make use of schema information, if available, which may improve the processing and results. Other interesting research could deal with the incorporation of tree similarity [15] and approximate queries [24] in both the object extraction and the object comparison phase.

7. REFERENCES

[1] Internet movie database: http://www.imdb.com/.

[2] Mondial database: http://www.dbis.informatik.uni-goettingen.de/mondial.

[3] Ananthakrishna, R., Chaudhuri, S., and Ganti, V. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB-2002)* (Hong Kong, China, 2002).

[4] Baeza-Yates, R. A., and Ribeiro-Neto, B. A. Modern information retrieval. *ACM Press / Addison-Wesley* (1999).

[5] Bartolini, I., Ciaccia, P., and Patella, M. String matching with metric trees using an approximate distance. In *Proceedings of the 9th International Symposium on String Precessing and Information Retrieval (SPIRE-2002)* (Belo Horizonte, Brazil, 2002), pp. 271–283.

[6] Bilenko, M., and Mooney, R. J. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)* (Washington, DC, 2003).

[7] Bilenko, M., and Mooney, R. J. Employing trainable string similarity metrics for information integration. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web* (Acapulco, Mexico, Aug. 2003), pp. 67–72.

[8] Galhardas, H., Florescu, D., Shasha, D., Simon, E., and Saita, C. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB-2001)* (Rome, Italy, 2001), pp. 371–380.

[9] Gravano, L., Ipeirotis, P., Jagadish, H., Koudas, N., Muthukrishnan, S., and Srivastava, D. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB-2001)* (Roma, Italy, 2001), pp. 491–500.

[10] Hernández, M. A., and Stolfo, S. J. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD-95)* (San Jose, CA, May 1995), pp. 127–138.

[11] Hylton, J. A. Identifying and merging related bibliographic records. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1996.

[12] Jaro, M. A. Probabilistic linkage of large public health data files. *Statistics in Medicine 14*, 5–7, 491–498.

[13] Jin, L., Li, C., and Mehrotra, S. Efficient similarity string joins in large data sets. Tech. Rep. TR-DB-02-04, UCI ICS, 2002.

[14] Jin, L., Li, C., and Mehrotra, S. Efficient record linkage in large data sets. In *Procceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA-03)* (Kyoto, Japan, 2003), pp. 137 –.

[15] Kailing, K., Kriegel, H.-P., Schnauer, S., and Seidel, T. Efficient similarity search for hierarchical data in large databases. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT-2004)* (Heraclion, Crete, 2004), pp. 676–693.

[16] Lim, E.-P., Srivastava, J., Prabhakar, S., and Richardson, J. Entity identification in database integration. In *Proceedings of the 9th International Conference on Data Engineering (ICDE-93)* (April 1993), pp. 294–301.

[17] Monge, A. E., and Elkan, C. P. The field matching problem: Algorithms and applications. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)* (Portland, OR, August 1996), pp. 267–270.

[18] Monge, A. E., and Elkan, C. P. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the SIGMOD 1997 Workshop on Research Issues on Data Mining and Knowledge Discovery* (Tuscon, AZ, May 1997), pp. 23–29.

[19] Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys, Volume 33, pages 31-88* (2003).

[20] Quass, D., and Starkey, P. Record linkage for genealogical databases. In *Proceedings of the KDD-2003 Workshop on Data Cleaning, Record Linkage, and Object Consolidation* (Washington, DC, 2003), pp. 40–42.

[21] Rahm, E., and Do, H. H. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin, Volume 23, pages 3-13* (2000).

[22] Raman, V., and Hellerstein, J. M. Potter's wheel: An interactive data cleaning system. In *Proceedings of 27th International Conference on Very Large Databases (VLDB-2001)* (Rome, Italy, 2001), pp. 381–390.

[23] Sarawagi, S., and Bhamidipaty, A. Interactive deduplication using active learning. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)* (Edmonton, Alberta, 2002).

[24] Schlieder, T. Schema-driven evaluation of approximate tree-pattern queries. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT-2002)* (Prague, Czech Republic, 2002), pp. 514–532.

[25] Winkler, W. E. Advanced methods for record linkage. Tech. rep., Statistical Research Division, U.S. Census Bureau, Washington, DC, 1994.

[26] Winkler, W. E. Data cleaning methods. In *Proceedings of the KDD-2003 Workshop on Data Cleaning, Record Linkage, and Object Consolidation* (Washington, DC, 2003), pp. 1–6.

[27] Yan, T. W., and Garcia-Molina, H. Duplicate removal in information dissemination. In *Proceedings of 21th International Conference on Very Large Data Bases (VLDB-95)* (Zurich, Switzerland, 1995), pp. 66–77.