

DEEPETECT: An Extensible System for Detecting Attribute Outliers & Duplicates in XML

Qiangfeng Peter Lau, Wynne Hsu, Judice L. Y. Koh, and Mong Li Lee

School of Computing
National University of Singapore
{plau, whsu, jkoh, leeml}@comp.nus.edu.sg

Abstract. XML, the eXtensible Markup Language, is fast evolving into the new standard for data representation and exchange on the WWW. This has resulted in a growing number of data cleaning techniques to locate “dirty” data (artifacts). In this paper, we present DEEPETECT – an extensible system that detects attribute outliers and duplicates in XML documents. Attribute outlier detection finds objects that contain deviating values with respect to a relevant group of objects. This entails utilizing the correlation among element values in a given XML document. Duplicate detection in XML requires the identification of subtrees that correspond to real world objects. Our system architecture enables sharing of common operations that prepare XML data for the various artifact detection techniques. DEEPETECT also provides an intuitive visual interface for the user to specify various parameters for preprocessing and detection, as well as to view results.

1 Introduction

Data overload, combined with the widespread use of automated large-scale analysis and mining, has led to the rapid depreciation of data quality. Data cleaning is an emerging domain that aims at improving data quality through the detection and elimination of artifacts. We define artifacts to comprise of errors, discrepancies, redundancies, ambiguities, and incompleteness that hamper the efficacy of analysis or data mining. Recent years have seen a rapid proliferation of semi-structured data models such as XML (eXtensible Markup Language) as a new standard for data representation and exchange on the WWW. Increasingly more databases are converted into XML formats to facilitate data access and integration, for example, the UniProt database of the worldwide protein sequences [1].

Despite the paradigm shift, development of data cleaning techniques for XML data is still at its infancy. Existing works mainly focus on duplicate detection in XML documents [2–4]. Outlier detection is also important in data cleaning, since outliers are often data noise or errors that diminish the accuracy of data mining. There are two types of outliers: class outliers and attribute outliers. A

class outlier is a multivariate data point (tuple) which does not fit into any class by definitions of distance, density, or nearest-neighbor. An attribute outlier is a univariate point which exhibits deviating correlation behavior with respect to other attributes [6]. This work focuses on attribute outliers, and we use the term outlier interchangeably with attribute outlier.

XML data differs from relational data in several aspects that limit the direct adaptation of conventional detection methods. Moreover, the hierarchical relationships between the XML elements provide additional contextual information to the detection problem. This enables the separation of the XML document into smaller workable partitions.

In light of the need to be able to find artifacts in XML data, we design DEEPDETECT as an extensible system for detecting both attribute outliers and duplicates in XML documents. Outlier detection finds *objects* that contain deviating values (outliers) with respect to a relevant group of objects [5, 6]. Duplicate detection in XML requires the identification of subtrees that corresponds to real-world objects for subsequent subtree matching.

In this paper, we describe the attribute outlier and duplicate detection capabilities of the DEEPDETECT system. DEEPDETECT is extensible – new artifact detection algorithms can be easily added based on our architecture. The system supports common preprocessing features and provides capabilities for the necessary correction of artifacts after detection. The graphical user interface of the system also makes it easy for novice users to configure and run the various artifact detection algorithms on XML documents.

2 Motivating Example

In this section, we use examples to illustrate the notions of attribute outliers and duplicates in XML documents.

2.1 Attribute Outliers in XML

We use the definitions in the XODDS framework [5] for the detection of attribute outliers in XML. We define an *object* as an instance of a related set of values that correspond to a real-world entity. For example, Figure 1 shows an XML document containing bank accounts and their corresponding transactions. A *Transaction* object consists of the values of the elements: *Amt*, *Type*, and *Bank*. The leftmost *Transaction* object, denoted as $Obj(Transaction)$, is the set $\{\langle Amt/\$30 \rangle, \langle Type/C \rangle, \langle Bank/YZ \rangle\}$.

The hierarchical structure of XML data serves to organize data according to their relevance – the closer the nearest shared ancestor is to the two elements, the more relevant they are, where closeness is measured by difference in ancestor-descendant element depth. In Figure 1, the *Transactions* elements serve to organize relevant *Transaction* elements that are made through the same *Account*. Likewise, the *Accounts* element serves to organize the relevant *Account*

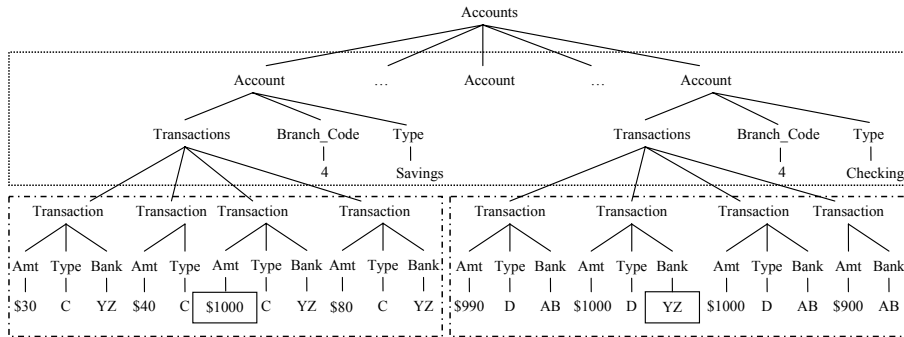


Fig. 1. Example of Bank Account XML and Object Groupings (correlated subspaces)

elements. We refer to elements such as *Transactions* and *Accounts* as *pivot* elements.

This notion of *pivot* elements allows partitioning of XML data into groups of objects according to their relevance. Each partition can be analyzed for attribute outliers independently, aiding in the scalability of the system. Furthermore, local outliers detected within a group of objects are more meaningful than outliers detected across all objects. Figure 1 illustrates three such relevant groups (boxed), one for *Account* objects and two for *Transaction* objects. When analyzing for transactional outliers, it is more useful to detect them within the same bank account than across all bank accounts. Hence transactions in the same bank account are grouped together. Such related groupings are referred to as *correlated subspaces* that are identifiable through the pivot elements.

An attribute outlier is a value in an object (the text of an XML element) that rarely occurs together with the other values in the same object within a particular correlated subspace. For example, the solid-boxed values $\langle \text{Amt}/\$1000 \rangle$ and $\langle \text{Bank}/\text{YX} \rangle$ in Figure 1 are outliers. $\langle \text{Amt}/\$1000 \rangle$, is an outlier because it is an unusually high transaction amount with respect to the values for *Type* and *Bank* compared to other transactions in the same account. $\langle \text{Bank}/\text{YX} \rangle$ is an outlier as the other transactions with the similar values for *Type* and *Amt* all involve $\langle \text{Bank}/\text{AB} \rangle$ instead. A subset of the elements for each transaction object is used to compute for outlier-ness and is referred to as the *correlated neighbors*.

Measuring outlier-ness involves counting the *supports* of correlated neighbors within correlated subspaces and comparing them. For example, the support of the correlated neighbor, $\{ \langle \text{Type}/\text{C} \rangle, \langle \text{Bank}/\text{YZ} \rangle \}$, of the correlated subspace given by the left *Transactions* element in Figure 1, is 3 since it appears thrice in the subspace. Two measures of outlier-ness were given in [5], namely the *xO-measure* and *xQ-measure* that measures outliers within correlated subspaces, whereby a smaller value indicates a higher degree of outlier-ness.

It is interesting to note that the outliers do not necessarily indicate error in data; they can be the result of suspicious activity that may have legal impli-

cations, as in the case of our Bank Account example. Hence, this is an added benefit of the ability to detect attribute outliers in a cleaning system.

2.2 Duplicates in XML

A real-world object (e.g. a movie, a CD, etc) can be mapped to a set of elements in an XML document. We call this set of elements an entity¹. Duplicate detection aims to find clusters of duplicate *entities* in the XML document.

For example, in Figure 1, if the intention is to detect duplicate transactions across all bank accounts, then we can map a transaction in the real world to the entity that comprises of the three XML elements: *Amt*, *Type*, and *Bank*. However, if the intent is to detect duplicate transactions within the same bank account, then the entity should include an additional XML element such as the unique account information, *Account_ID* (not shown in Figure 1), to discriminate against transactions in other accounts. Hence, depending on the application, a domain expert is needed to identify the entity in the XML documents.

Having identified the entities, similarities between entities are computed as follows. We first use edit distance to determine the similarity between the values of each corresponding pair of elements in the entity. Two elements are similar if their edit distance is below some user-defined threshold. Next, we apply a weighted sum of the element pair similarities to compute the overall similarity between a pair of entities. If the overall similarity is below some threshold, then we can conclude that they are likely to refer to the same real-world object.

3 Architecture

In this section, we describe the main components of the DEEPDETECT system. As shown in Figure 2, DEEPDETECT comprises of a GUI module, a Specification module, a Data Preparation module and an Artifact Detection module.

3.1 GUI Module

This module consists of two main viewer components, the *XML Structure Viewer* and the *XML Editor*. The *XML Structure Viewer* is a tree-like viewer that allows the users to explore the XML structure. The viewer also collects user specification for discretization and input parameters required for the various artifact detection tasks. The *XML Editor* displays the detected artifacts' locations and allows the user to specify corrections (if necessary). Furthermore, it also displays the abbreviated XPath's of the XML elements related to the artifacts so that the user may use external applications to query specific portions of the XML document.

¹ Note that an entity, like an object, is an embodiment of a real-world object.

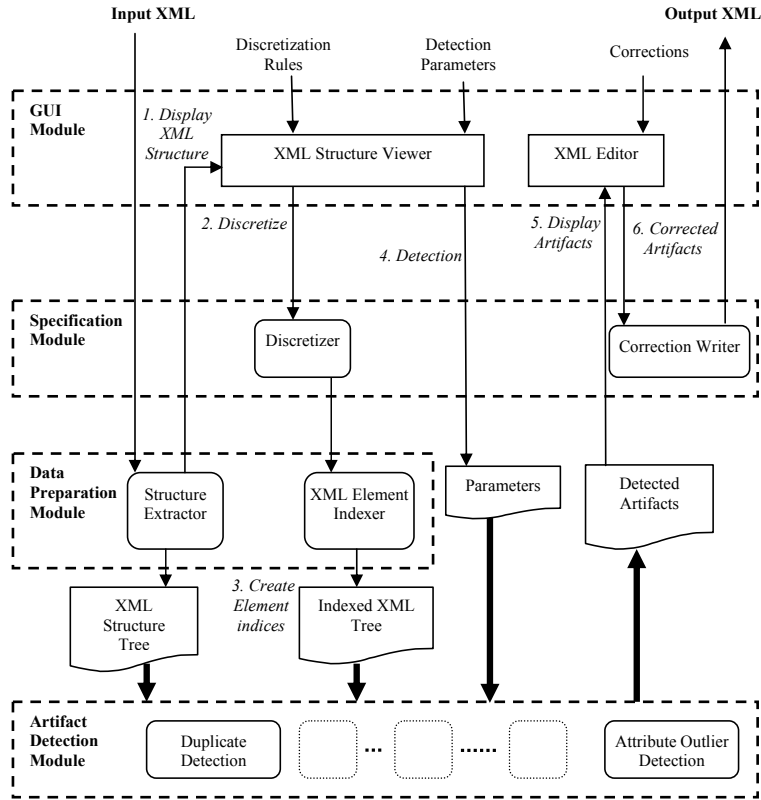


Fig. 2. DEEPDETECT System Architecture

3.2 Specification Module

This module encompasses the input and output specification for the data cleaning process. The *Discretizer* groups the input numerical values according to the discretization interval width as specified by the user. The transformed XML data is used as input to the various artifact detection processes. The *Correction Writer* is responsible for exporting the output XML document with user corrections that were previously specified through the XML editor interface.

3.3 Data Preparation Module

The data preparation module converts the XML document to an internal representation that facilitates the detection of artifacts. The *Structure Extractor* automatically processes the XML document to build a structure tree and determine the multiplicity of the elements. This can be achieved through a depth-first traversal of the XML tree. A node is created for each unique element name. Each node has a hash table that provides fast access to elements with the same name.

The result is a compact structure tree that describes the nesting of XML elements as shown in Figure 3. In the process, “one” or “many” multiplicity of the XML elements are also identified.

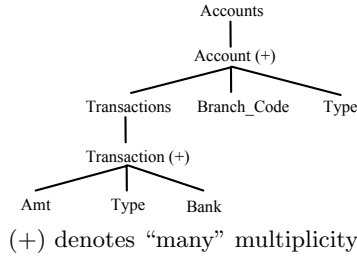


Fig. 3. Bank Account XML Structure Tree

The *XML Element Indexer* uses the interval-based labeling scheme to convert the XML document to an internal indexed representation for faster lookup during processing. XML elements are divided into different files based on the element names. A depth-first traversal is performed on the XML tree starting from the root. In each XML element’s index, the *start* and *end* order of traversal is recorded as a traversal interval pair (s, e). For example, in Figure 4, starting from the root element, *Accounts*, the *Transaction* element will have the interval (4, 11), while its children, $\langle \text{Amt}/\$30 \rangle$, $\langle \text{Type}/C \rangle$, and $\langle \text{Bank}/YZ \rangle$ will have the intervals (5, 6), (7, 8), and (9, 10) respectively. Note that the intervals for the descendant nodes are always bounded by the ancestors’ intervals. In addition, during indexing, the element depth and the abbreviated XPath is also determined and stored. With this, we can maintain the order of appearance of the XML elements in the original XML document.

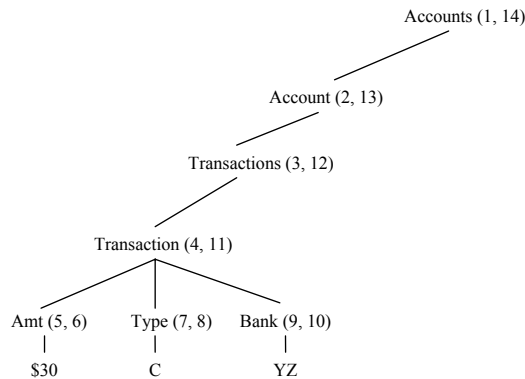


Fig. 4. An example Bank Account XML with traversal interval pair labels

3.4 Artifact Detection Module

This module contains the key components for detecting the various artifacts. The architecture is extensible with additional detection routines which may make use of the same specification and data preparation capabilities.

The *Attribute Outlier Detection* component projects correlated subspaces and does support counting before applying a user specified metric to evaluate outliers. The details of this component is discussed in Section 4.

The *Duplicate Detection* component extracts the duplicate candidates from user specified entities. Candidate entities are compared to identify duplicate clusters. The result of this process is a list of detected duplicate clusters that the user may choose to eliminate from the output XML document. The details of this component are discussed in Section 5.

4 Attribute Outlier Detection

The attribute outlier detection component finds outliers in XML data in the steps according to the architecture shown in Figure 5 that is based on the XODDS framework presented in [5, 8]. We use the XML structure tree to automatically identify XML elements having leaf elements as their children. These XML elements are objects. For example, from Figure 3, *Account* and *Transaction* elements will be immediately identified as object elements.

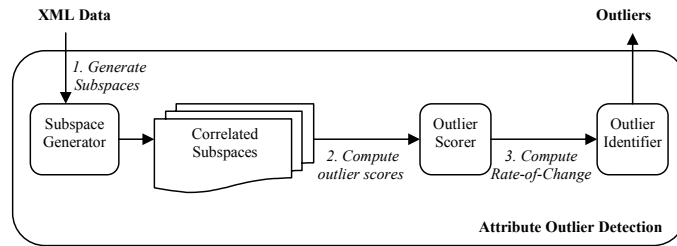


Fig. 5. Outlier Detection Architecture

The *Subspace Generator* extracts the objects into their corresponding correlated subspaces. To do so, pivot elements must be identified. Given the XML structure tree, for a particular object, we can identify the name of the element that serves as the pivot element as follows:

1. Use the XML structure tree to find a path, P , from an object element v to the root element r .
2. Suppose we have the path $P = v_0, \dots, v_n$ where $v_0 = v$ is the object element and $v_n = r$ is the root element. Then the pivot element of the correlated subspace for v is the first occurrence of v_p , $1 \leq p \leq n$, s.t. v_{p-1} has an element multiplicity of “many”.

Using the method above, in Figure 3, we can identify *Transactions* as the pivot of *Transaction* and *Accounts* as the pivot of *Account*. Once the pivot elements are identified, grouping objects into correlated subspaces is a matter of placing all the elements that are descendants of the pivot element into the same subspace.

Once the correlated subspaces are generated, for each subspace, projections are made over the objects to generate their correlated neighbors. For example, the leftmost transaction object, $\{\langle \text{Amt}/<100 \rangle, \langle \text{Type}/C \rangle, \langle \text{Bank}/YZ \rangle\}$ in Figure 1 will have 7 correlated neighbors, namely:

1. $\{\langle \text{Amt}/<100 \rangle\}$,
2. $\{\langle \text{Type}/C \rangle\}$,
3. $\{\langle \text{Bank}/YZ \rangle\}$,
4. $\{\langle \text{Amt}/<100 \rangle, \langle \text{Type}/C \rangle\}$,
5. $\{\langle \text{Amt}/<100 \rangle, \langle \text{Bank}/YZ \rangle\}$,
6. $\{\langle \text{Type}/C \rangle, \langle \text{Bank}/YZ \rangle\}$,
7. $\{\langle \text{Amt}/<100 \rangle, \langle \text{Type}/C \rangle, \langle \text{Bank}/YZ \rangle\}$.

The support of each neighbor is counted in the process.

From the support values, the *Outlier Scorer* computes the degree of outlier-ness, using the measures given in [5], as a *score*. Values that occur frequently are less likely to be outliers. Thus, to reduce score computation, these values are filtered when their counts are greater than a user specified *min_sup* threshold.

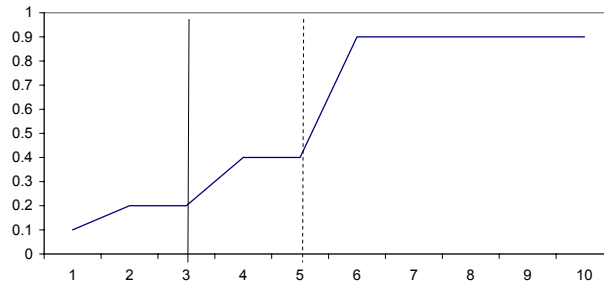


Fig. 6. Plot of potential Outlier index versus score. A situation whereby rate-of-change admits too many false positives

After the score for each potential outlier is calculated, the *Outlier Identifier* sorts them and an outlier threshold is automatically determined by computing the largest *rate-of-change* (difference between two adjacent scores). Note that there are cases where, due to the natural occurrences in the XML document, the *min_sup* threshold parameter used in the scorer fails to remove enough false positives. A hypothetical situation is shown in Figure 6 that plots the potential outliers versus score. Computing the rate-of-change will give a cut-off threshold at the 5th potential outlier (dotted line) that might admit too many false positives. Such situations can be remedied by introducing a “soft_cut”

parameter, $k \in (0, 1]$, that only considers the the top $k \times n$ out of n potential outliers when computing the rate of change. For example, setting $k = 0.5$ for the situation in Figure 6 will first discard the 6th to 10th potential outliers. Then, the resulting rate-of-change will give a cut-off threshold at the 3rd potential outlier.

Potential outliers with scores below the cut-off threshold are marked as outliers. Finally, objects that contain these outliers are returned to the front-end of the system to be highlighted to the user.

5 Duplicate Detection

The duplicate detection component is shown in Figure 7. An *entity definition* is a mapping of a real-world object to a set of XML element names as well as a set of thresholds, Θ . A *tuple* is a pair of element name and its corresponding value. An *entity* consists of a set of tuples that conforms to an entity definition. For example, in Figure 1, using the definition of a transaction entity given in Table 1, the leftmost transaction entity comprises of the set of tuples $\{\langle \text{Amt}, \$30 \rangle, \langle \text{Type}, C \rangle, \langle \text{Bank}, YZ \rangle\}$.

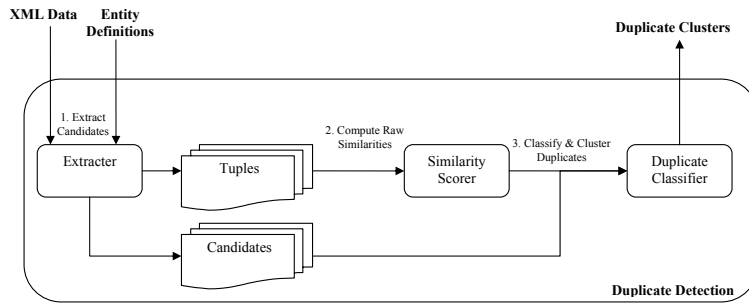


Fig. 7. Duplicate Detection Architecture

Element	Threshold
Amt	0.5
Type	0.5
Bank	0.4

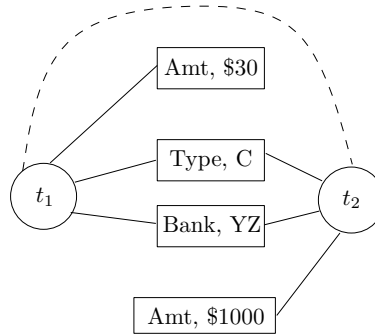
Table 1. Example Entity Definition for Bank Transactions

The duplicate detection process begins with the XML data and entity definitions as input. The *Extractor* module extracts entities in the XML documents that correspond to the entity definitions. These are the duplicate candidates. In

the process, the Extractor also generates the unique tuples from the candidate entities. Note that the duplicate candidates are extracted from the internally indexed XML document instead of the original document. This allows more direct access to the XML data through the indices created by the XML Element Indexer.

The *Similarity Scorer* computes the pair-wise similarity between the text strings in tuples belonging to the same XML element based on their edit distance (Levenshtein) normalized to the length of the longer string. Each $\theta \in \Theta$ specified in the entity definition that corresponds to an XML element is used as a threshold to filter dissimilar pairs of tuples.

The *Duplicate Classifier* uses the tuples of candidate entities to build a directed graph [7]. This facilitates the direct access of the tuples of an entity. Figure 8 illustrates an example sub-graph of two entities from the graph used for computing duplicate entities. Vertices t_1 and t_2 refer to the first and third leftmost Transaction entities from Figure 1. Note that vertices representing tuples $\langle \text{Type}, \text{C} \rangle$ and $\langle \text{Bank}, \text{YZ} \rangle$ are connected to both t_1 and t_2 since these entities share the exact same unique tuples.



Note that a line with no arrowheads indicate bi-directional edge. The dotted line denotes a special edge connecting duplicate entities

Fig. 8. Example sub-graph of two Transaction entities

If two tuples are similar (as scored previously), directed edges are added from one tuple vertex to the entity vertices adjacent to the other tuple vertex. For example, in Figure 8, if tuples $\langle \text{Amt}, \$30 \rangle$ and $\langle \text{Amt}, \$1000 \rangle$ are scored as being similar, the edges $(\langle \text{Amt}, \$30 \rangle, t_2)$ and $(\langle \text{Amt}, \$1000 \rangle, t_1)$ will be added. In this way, an entity's similarity with another may be computed based on the number of paths from the entity vertex, through one tuple vertex to the other entity vertex. Examples of such paths from t_1 to t_2 are: $t_1 \rightarrow \langle \text{Type}, \text{C} \rangle \rightarrow t_2$, and $t_1 \rightarrow \langle \text{Bank}, \text{YZ} \rangle \rightarrow t_2$.

Once a pair of entities, u, v , are detected to be duplicates, a special duplicate edge (u, v) is added to the graph. For example, assuming that t_1 and t_2 qualify as duplicates in Figure 8, the dotted line represents a duplicate edge. Computing

the transitive closure over the subgraph formed by vertices linked by duplicate edges gives the duplicate clusters. These duplicate clusters are returned to the front-end of the system to be highlighted to the user.

6 Implementation

DEEPPDETECT is implemented in JAVA version 1.5. In order to handle large XML documents, DEEPPDETECT is implemented using the Xerces SAX XML parser [9] instead of the DOM parser as the DOM tree consumes too much main memory. The SAX parser returns a series of events (e.g. encountering the start tag of an element) when parsing to a event handler instead of returning a data-structure. We use an event chaining mechanism in which one event handler passes event information to another handler so that multiple operations can be handled in a single parse of the XML document. This is shown schematically in Figure 9. Note that a handler may fork the events to two separate handlers as in the case of Handler 1.

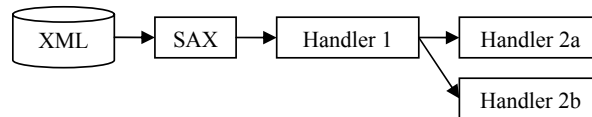


Fig. 9. Example event handler chaining

When loading a new XML document, DEEPPDETECT is able to discern an XML's structure. The XML file is copied to a separate storage directory to ensure subsequent modifications to it will not affect the original data. The copy step is done through the parser, thus allowing validation of the XML file. This is an example where copying and discerning the XML's structure can be done in a single pass using event handler chaining.

One common operation when detecting artifacts is counting the occurrences of data. Although hash tables are commonly used to do so, for large XML documents, unique data may require hash tables that are too big to fit in main memory. In such cases, a combination of size bounded hash tables and multi-pass approaches over the XML data is used.

For the implementation of the Outlier Detection component, there is the possibility of combinatorial explosion for high arity objects when data is sparse in relation to the size of the XML document. Hence projections on subspaces are done incrementally with intermediate results written to disk at the expense of computation time. To reduce the space consumed on disk, standard text compression is used.

7 Features

Figure 10 displays the preprocessing screen of DeepDetect on a sample XML document of a Compact Disc (CD) catalog. The user is presented with the XML tree structure. We provide a point and click interface for the user to specify the exact position in the tree structure to apply discretization rules in this preprocessing step. The values of elements that correspond to the specified element names will be discretized. For example, Figure 10 shows that the PRICE elements will be discretized using the bin width of 1.

To begin detecting artifacts, we provide the user with a unified configuration screen to choose which artifact detection algorithms to employ as shown in Figure 11.

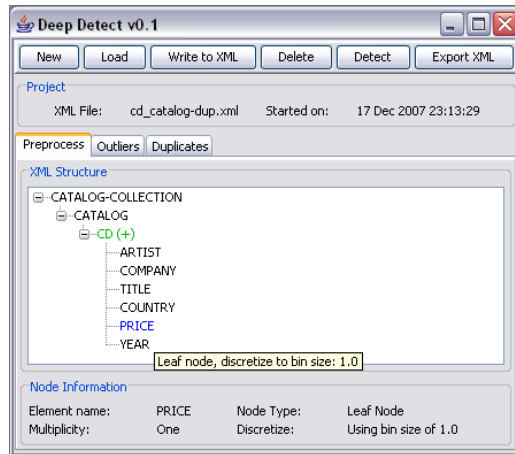


Fig. 10. Structure Tree View

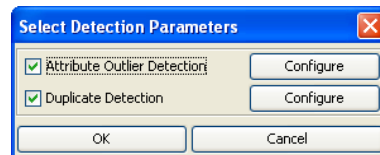


Fig. 11. Choosing which detection algorithms to include

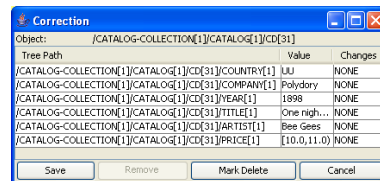


Fig. 12. Making corrections

Attribute outlier detection is primarily conducted by a number of back-end processes with minimal user interaction. The user specifies the required parameters such as *min_sup* and the soft-cut through the unified configuration screen.

Duplicate detection requires users to specify the entity definitions in detail. Hence we provide automatic creation of entity definitions by choosing entities based on elements with leaf elements as children. These entity definitions may then be customized (or removed) to suit the user's requirements. Figure 13 shows editing an example definition of a CD entity that contains all its children elements.

Once the required parameters for the chosen artifact detection algorithms have been entered, the system discretizes and builds an internal indexed representation of the XML document. This is followed by applying the various detection processes to identify outliers and duplicates.

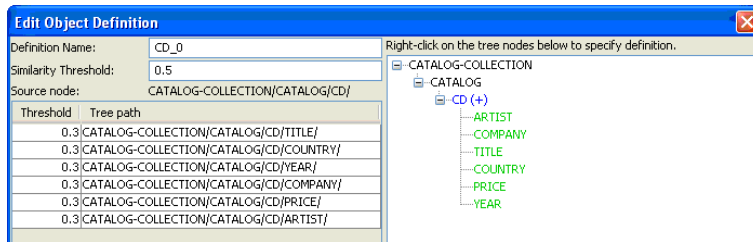


Fig. 13. Customizing entity definition for duplicate detection

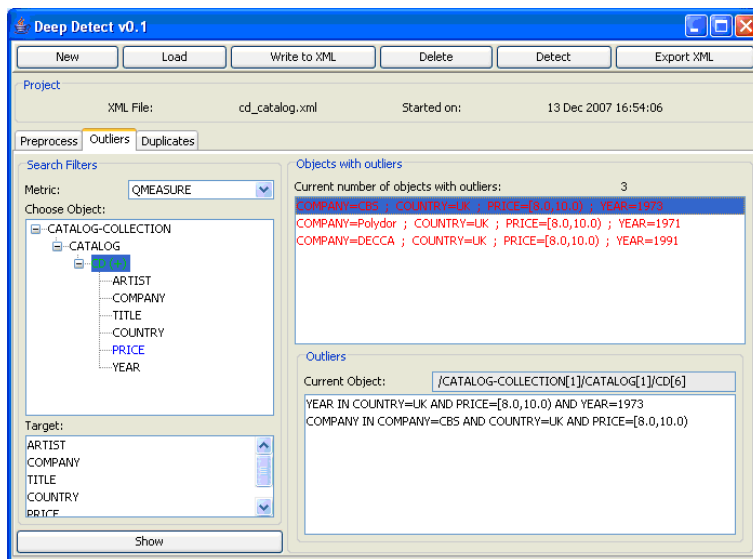


Fig. 14. Viewing identified objects that contain outliers

After detection, objects identified to contain attribute outliers may be browsed by the user. For each such object, the outliers are displayed together with the projection on the object that they are detected in. Figure 14 shows some highlighted CD objects with outliers. In particular the current CD object has $\langle \text{YEAR}/1973 \rangle$ as an outlier in $\{ \langle \text{COUNTRY}/\text{UK} \rangle, \langle \text{PRICE}/[8,10) \rangle, \langle \text{YEAR}/1973 \rangle \}$ which may indicate that most CDs in the data set are not priced at the range between [8, 10) in the UK during 1973. The unique XML element keys for each object, similar to XPath, is also displayed for identification.

Next, the duplicate clusters identified are also highlighted to the user. Figure 15 illustrates two duplicate CD entities. Note that they both have exactly the same tuples for $\langle \text{ARTIST}, \text{Bee Gees} \rangle$ and $\langle \text{TITLE}, \text{One night} \rangle$ only. Furthermore, their tuples for elements COMPANY, PRICE, COUNTRY, and YEAR are highly similar.

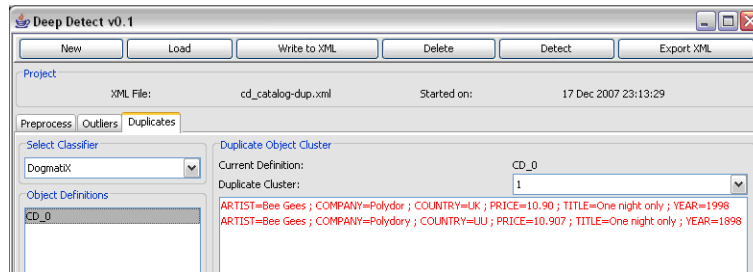


Fig. 15. Viewing duplicate clusters

The detected artifacts and their related XML elements and abbreviated XPathS are shown to the user through the Correction Writer (see Figure 12). Corrections made are saved and written when the XML file is exported. Alternatively the user may use the XPath information provided with other XML querying systems to make corrections.

8 Conclusion

In conclusion, we have presented the architecture of an extensible system, DEEP-DETECT, that provides common components for detecting the various forms of artifacts or dirty data in XML documents. These common components provide preprocessing, indexing of XML data and correction capabilities. An overview of how attribute outliers in XML data are identified based on the XODDS framework is given. We also described how duplicates are detected. The detected artifacts are mapped to the XML elements in the original XML document for identification. We provided a brief insight into the handling of large XML documents. Our graphical interface facilitates the specifications for preprocessing, artifact detection, viewing of detected artifacts, and data correction by a novice user.

References

1. Apweiler, R., Bairoch, A., Wu, C., Barker, W., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., et al., M.M.: UniProt: the universal protein knowledge-base. *Nucleic Acids Res.* **32** (2004) D115–D119
2. Low, W.L., Tok, W.H., Lee, M.L., Ling, T.W.: Data cleaning and XML: The DBLP experience. In: *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, Washington, DC, USA, IEEE Computer Society (2002) 269
3. Puhlmann, S., Weis, M., Naumann, F.: XML duplicate detection using sorted neighborhoods. In: *EDBT*. (2006) 773–791
4. Weis, M., Naumann, F.: Dogmatix tracks down duplicates in XML. In: *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, New York, NY, USA, ACM Press (2005) 431–442

5. Koh, J., Lee, M., Hsu, W., Ang, W.T.: Correlation-based attribute outlier detection in XML. In: Proceedings of the 24th International Conference on Data Engineering, Cancun, Mexico (2008)
6. Koh, J., Lee, M., Hsu, W., Lam, K.: Correlation-based detection of attribute outliers. In: 12th International Conference on Database Systems for Advanced Applications, Bangkok, Thailand (2007)
7. Weis, M., Naumann, F.: Detecting duplicate objects in xml documents. In: IQIS '04: Proceedings of the 2004 international workshop on Information quality in information systems, New York, NY, USA, ACM (2004) 10–19
8. Koh, J., Lee, M., Hsu, W., Ang, W.T.: Correlation-based attribute outlier detection in XML. Technical Report TRB6/07, National University of Singapore (2007)
9. <http://xerces.apache.org/>: Xerces Simple API for XML Parser