Ingredients for Accurate, Fast, and Robust XML Similarity Joins

Leonardo Andrade Ribeiro[1]* and Theo Härder[2]

[1] Department of Computer Science,
Federal University of Lavras, Brazil
laribeiro@dcc.ufla.br

[2] AG DBIS, Department of Computer Science,
University of Kaiserslautern, Germany
haerder@cs.uni-kl.de

Abstract. We consider the problem of answering similarity join queries on large, non-schematic, heterogeneous XML datasets. Realizing similarity joins on such datasets is challenging, because the semi-structured nature of XML substantially increases the complexity of the underlying similarity function in terms of both effectiveness and efficiency. Moreover, even the selection of pieces of information for similarity assessment is complicated because these can appear at different parts among documents in a dataset. In this paper, we present an approach that jointly calculates textual and structural similarity of XML trees while implicitly embedding similarity selection into join processing. We validate the accuracy, performance, and scalability of our techniques with a set of experiments in the context of an XML DBMS.

Keywords: XML, Similarity Join, Fuzzy Duplicate Identification

1 Introduction

As XML continues its path to becoming the universal information model, large-scale XML repositories proliferate. Very often, such XML repositories are non-schematic, or have multiple, evolving, or versioned schemas. In fact, a prime motivation for using XML to directly represent pieces of information is the ability of supporting ad-hoc or "schema-later" settings. For example, the flexibility of XML can be exploited to reduce the upfront cost of data integration services, because documents originated from multiple sources can be stored without prior schema reconciliation and queried afterwards in a best-effort manner.

Of course, the flexibility of XML comes at a price: loose data constraints can also lead to severe data quality problems, because the risk of storing inconsistent and incorrect data is greatly increased. A prominent example of such problems is the appearance of the so-called *fuzzy duplicates*, i.e., multiple and non-identical representations of a real-world entity [1]. Complications caused by such redundant information abound in common business practices. Some examples are misleading results of decision support queries due to erroneously inflated

* Work done at the University of Kaiserslautern, Germany.

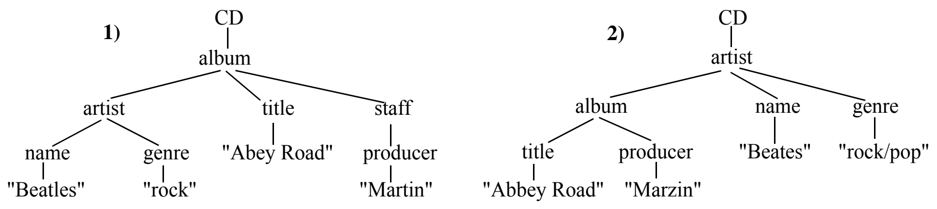


Fig. 1: Heterogeneous XML data

statistics, inability of correlating information related to the same customer, and unnecessarily repeated operations, e.g., mailing, billing, and leasing of equipment. For relational data, fuzzy duplicate identification is often needed on text fields owing to misspellings and naming variations. For XML — commonly modeled as a labeled tree —, this task is even more critical, because also structure, in addition to text, may present deviations. For instance, consider the sample data from music inventory databases shown in Fig. 1. Subtrees **1)** and **2)** in Fig. 1 apparently refer to the same CD. However, the use of conventional operators based on exact matching to group together such duplicate data is futile: subtree **1)** is arranged according to `album`, while subtree **2)** is arranged according to `artist`, there are extra elements (`staff` in subtree **1)**), and several typos in the content of text nodes (e.g., "Beatles" and "Beates").

In this paper, we present the design and evaluation of an approach to XML similarity joins. In Sect. 2, we present the main ingredients of our approach, which are the decomposition of the computation into three components that can be independently dealt with and a strategy for delimiting textual and structural representations based on XML path similarity. We formally define the logical similarity join operator, which implicitly incorporates similarity selection as sub-operation in Sect. 3. Then, in Sect. 4, we demonstrated that our approach can be applied flexibly to assess the similarity of ordered and unordered trees, deal with large datasets, and deliver accurate results with a set of experiments in the context of an XML DBMS. Related work is discussed in Sect. 5, before we wrap up with the conclusions in Sect. 6.

2 Main Ingredients

2.1 Similarity Functions

We focus on the class of token-based similarity functions, which ascertains the similarity between two entities of interest by measuring the overlap between their set representations. We call such set representation the *profile* of an entity, the elements of the profile are called *tokens*; optionally, a *weighting scheme* can be used to associate weights to tokens. Token-based similarity functions allow measuring textual and structural similarity in a unified framework and provide a very rich similarity space by varying profile generation methods, weighting schemes, set similarity functions, or any combination thereof. Moreover, these functions are computationally inexpensive and we can leverage a wealth of techniques for

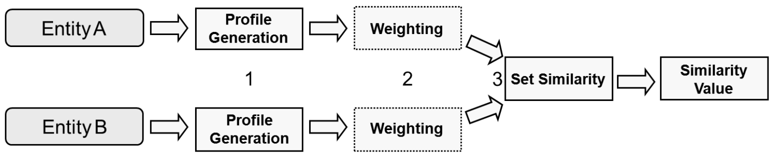


Fig. 2: Evaluation of token-based similarity functions

similarity joins on strings (see [2] and references therein). Fig. 2 illustrates the three main components of token-based similarity functions and the evaluation course along them towards a similarity value.

Profile Generation The profile of an entity is generated by splitting its representation into a set of tokens; we call this process *tokenization*. The idea behind tokenization is that most of the tokens derived from significantly similar entities should agree correspondingly. For XML, tokenization can be applied to text, structure, or both. We next describe methods capturing text and structure in isolation; methods that generate tokens conveying both textual and structural information are employed in Sect. 4.

A well-known textual tokenization method is that of mapping a string to a set of *q-grams*, i.e., substrings of size q. For example, the *2-gram* profile of the string "Beatles" is {'Be', 'ea', 'at', 'tl', 'le', 'es'}. Structural tokenization methods operate on element nodes capturing labels[3] and relationships. A simple structural (path) tokenization method consists of simply collecting all element node labels of a path. Thus, the profile of the path `/CD/album/artist/name` would be {'CD', 'album', 'artist', 'name'}. Note that, as described, the result of both tokenization methods could be a multi-set. We convert a multi-set to sets by concatenating the symbol of a sequential ordinal number to each occurrence of a token. Hence, the multi-set {'a','b','b'} is converted to {a$\circ$1, b$\circ$1, b$\circ$2} (the symbol $\circ$ denotes concatenation).

Weighting Schemes In many domains, tokens show non-uniformity regarding some semantic properties such as discriminating power. Therefore, the definition of an appropriate weighting scheme to quantify the relative importance of each token for similarity assessment is instrumental in obtaining meaningful similarity results. For example, the widely used *Inverse Document Frequency* (*IDF*) weights a token t as follows: $IDF(t)=ln(1 + N/f_t)$, where f_t is the frequency of token t in a database of N documents. The intuition of IDF is that rare tokens usually carry more content information and are more discriminative. Besides statistics, other kinds of information can be used to calculate weights.

[3] We assume that element labels are drawn from a common vocabulary. Semantic integration of vocabulary (or ontology) is a closely related but different problem from similarity matching of structure and textual content, which is our focus here.

The Level-based Weighting Scheme (*LWS*) [3] weights structural tokens according to node nesting depth in a monotonically decreasing way: given a token t derived from a node at nesting level i, its weight is given by $LWS(t) = e^{\beta i}$, where $\beta \leq 0$ is a decay rate constant. The intuition behind *LWS* is that in tree-structured data like XML more general concepts are normally placed at lower nesting depths. Hence, mismatches on such low-level concepts suggest that the information conveyed by two trees is semantically "distant".

Set Similarity Tokenization delivers an XML tree represented as a set of tokens. Afterwards, similarity assessment can be reduced to the problem of set overlap, where different ways to measure the overlap between profiles raise various notions of similarity. In the following, we formally define the *Weighted Jaccard Similarity*, which will be used in the rest of this paper. Several other set similarity measures could however be applied [2].

Definition 1. *Let $\mathcal{P}_1$ be a profile and $w(t, \mathcal{P}_1)$ be the weight of a token t in $\mathcal{P}_1$ according to some weighting scheme. Let the weight of $\mathcal{P}_1$ be given by $w(\mathcal{P}_1) = \sum_{t \in \mathcal{P}_1} w(t, \mathcal{P}_1)$. Similarly, consider a profile $\mathcal{P}_2$. The Weighted Jaccard Similarity between $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as $WJS(\mathcal{P}_1, \mathcal{P}_2) = \frac{w(\mathcal{P}_1 \cap \mathcal{P}_2)}{w(\mathcal{P}_1 \cup \mathcal{P}_2)}$, where $w(t, \mathcal{P}_1 \cap \mathcal{P}_2) = min(w(t, \mathcal{P}_1), w(t, \mathcal{P}_2))$.*

Example 1. Consider the profiles $\mathcal{P}_1$ =$\{\langle$'Be', 5$\rangle$,$\langle$'ea', 2$\rangle$,$\langle$'at', 2$\rangle$,$\langle$'tl', 2$\rangle$,$\langle$'le', 1$\rangle$,$\langle$'es', 4$\rangle\}$ and $\mathcal{P}_2$ =$\{\langle$'Be', 5$\rangle$,$\langle$'ea', 2$\rangle$,$\langle$'at', 2$\rangle$,$\langle$'te', 1$\rangle$,$\langle$'es', 4$\rangle\}$ — note the token-weight association, i.e., $\langle$t, $w(t)\rangle$. Therefore, we have $WJS(\mathcal{P}_1, \mathcal{P}_2) \approx 0.76$.

2.2 XML Path Clustering

There are several challenges to realizing similarity joins on heterogeneous XML data. Regarding effectiveness, structural and textual similarities have to be calculated and combined. Text data needs to be specified, because often only part of the available textual information is relevant for similarity matching, and can be only *approximately* selected, because the underlying schema is unknown or too complex. Regarding efficiency, it is important to generate compact profiles and avoid repeated comparisons of structural patterns that may appear many times across different XML documents. Next, we briefly review our approach based on path clustering, which provides the basis for tackling all the issues above. For a detailed discussion, please see [3].

Our approach consists of clustering all path classes of an XML database in a pre-processing step. Path classes uniquely represent paths occurring at least once in at least one document in a database. The similarity function used in the clustering process is defined by the path tokenization method and the LWS weighting scheme described earlier and some set similarity function like WJS. As a result, we obtain the set $PC = \{pc^1, \ldots, pc^n\}$, where pc^i is a cluster containing similar path classes and i is referred to as *Path Cluster Identifier* (PCI). Given a path p appearing in some document, we say that $p \in pc^i$ iff the path class of p

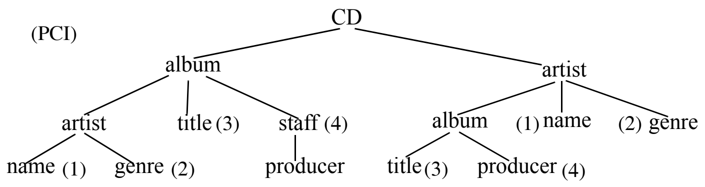


Fig. 3: Path synopsis annotated with PCI values

is in pc^i. Further, we can reduce similarity matching between paths to a simple equality comparison between their corresponding PCIs, because the actual path comparison has already been performed during the clustering process.

Prior to clustering, all path classes of a database have to be first collected. This can be done in a single pass over the data. Preferably, we can use the so-called *Path Synopsis* (PS), a tree-structured index providing and maintaining a structural summary of an XML database [4]. Each node in a PS represents a (partial) path class. Fig. 3 depicts the PS of the sample database shown in Fig. 1 annotated with PCI values, where similar paths have the same PCI.

PCI values are used to guide the selection of text data that will compose the textual representation of an entity. For this, we define the set $PC_t \subseteq PC$: only text nodes appearing under a path in PC_t are used to generated tokens conveying textual information. We let users specify the PC_t set by issuing simple *path queries* like `/a/b/c`, which are approximately matched against the elements of PC. The K path clusters with highest similarity to each path query are selected to form PC_t. To enable very short response times, path clusters are represented by a single *cluster representative*, to which path queries are compared, and implemented as little memory-resident inverted lists. The PC_t can be interactively or automatically constructed, in which path queries are embedded into the main similarity join query. In the following, we assume that PC_t is given.

3 Tree Similarity Join

We are now ready to define our Tree Similarity Join (TSJ) operator. This operator takes as input two XML databases and outputs all pairs of XML trees whose similarity is greater than a given threshold.

Definition 2. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two XML databases and $exp(\mathcal{D})$ be an XPath or XQuery expression over a database $\mathcal{D}$. Further, let tok be a tokenization method that, given a set PC_t of PCIs, maps an XML tree T to a profile $tok[PC_t](T)$, ws be a weighting scheme that associates a weight to every element of a given input set, and ss be a set similarity measure. Let sf be the similarity function defined by the triple $\langle tok[PC_t], ws, ss\rangle$, which returns the similarity between two XML trees T_1 and T_2, $sf(T_1, T_2)$ as a real value in the interval $[0, 1]$. Finally let τ be a similarity threshold, also in the interval $[0, 1]$. The Tree Similarity Join between the tree collections specified by $exp_1(\mathcal{D}_1)$ and $exp_2(\mathcal{D}_2)$, denoted by $TSJ(exp_1(\mathcal{D}_1), exp_2(\mathcal{D}_2), sf, \tau)$, returns all scored tree pairs $\langle(T_1, T_2), \tau\prime\rangle$ s.t. $(T_1, T_2) \in exp_1(\mathcal{D}_1) \times exp_2(\mathcal{D}_2)$ and $sf(T_1, T_2) = \tau\prime \geq \tau$.*

Note that we can evaluate TSJ over the same database by specifying $\mathcal{D}_1 = \mathcal{D}_2$ or over a single XML tree collection by specifying $exp_1(\mathcal{D}_1)=exp_2(\mathcal{D}_2)$ or simply omitting the second parameter (hence, defining a self-similarity join).

The course of the TSJ evaluation closely follows that of token-based similarity functions shown in Fig. 2. A pre-step consists of accessing and fetching the trees into a memory-resident area, forming the input of TSJ. To this end, we can fully leverage the query processing infrastructure of a host XML DBMS environment to narrow the similarity join processing to the subset of XML documents (or fragments thereof) specified by the query expression. The next steps, **1)** Profile Generation, **2)** Weighting, and **3)** Set Similarity can be independently implemented and evaluated in a pipelined fashion. Profile Generation produces tokens capturing only structure and structure in conjunction with text. As general strategy, text data appearing under a path in PC_t is converted to a set of *q-grams* and appended to the corresponding structural tokens. Note that, because the set PC_t is obtained by similarity matching between path queries and the elements of PC (recall Sect. 2.2), this strategy implicitly embeds similarity selection into the join processing. The realization of Weighting is straightforward. For certain weighting schemes, such as IDF, we need the frequency of all tokens in the database. We can store and easily maintain this information in a simple memory-resident token-frequency table. (Assuming four bytes each for the hashed token value and its frequency, 1.3 million tokens would require around 10MB memory space, which is hardly an issue with modern computers.) Set Similarity is implemented by the set similarity join algorithm based on inverted lists presented in [2]. This algorithm requires sorting the tokens of each profile in increasing order of frequency in the data collection as well as sorting the profiles in increasing order of their size. The sorting of tokens is done in step **2)** using the token-frequency table, while we only need an additional sort operator to deliver sorted profiles to step **3)**.

The TSJ evaluation is completely done “on-the-fly”, i.e, we do not rely on any indexing scheme to provide access or maintenance on pre-computed profiles. Besides avoiding the issue of index update in the presence of data updates, steps **2)** and **3)** do not take an exceedingly large fraction of the overall processing time. Finally, we note that on-the-fly evaluation is the only option in virtual data integration scenarios where the integrated data is not materialized.

4 Experiments

The objectives of our empirical experiments are to measure accuracy of the similarity functions (i), overall performance of the TSJ operator (ii), relative performance of the TSJ components (iii), their scalability (iv), and to compare the performance of TSJ using different similarity functions.

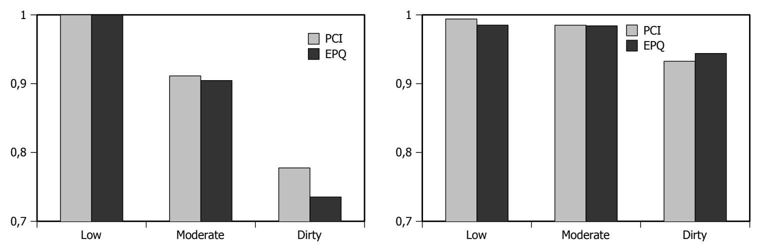
We used two real-world XML databases on protein sequences: *SwissProt* (http://us.expasy.org/sprot/) and *PSD* (http://pir.georgetown.edu/). We deleted the root node of each XML dataset to obtain sets of XML documents. The resulting documents are structurally very heterogeneous. On average, *SwissProt* has a larger number of distinct node labels and exhibits larger and wider trees. We defined the set PC_t by issuing two path queries for each dataset: `/Ref/Author` and `Org` on *SwissProt* and `organism/ formal` and `sequence` on *PSD*. The resulting text data on *PSD* is about 2x larger than on *SwissProt*.

Using these datasets, we derived variations containing fuzzy duplicates by creating exact copies of the original trees and then performing transformations, which aimed at simulating typical deviations between fuzzy duplicates appearing in heterogeneous datasets and those resulting from schema evolution, text data entry errors, and the inherent structural heterogeneity that naturally emanates from the XML data model. Transformations on text nodes consist of word swaps and character-level modifications (insertions, deletions, and substitutions); we applied 1–5 such modifications for each dirty copy. Structural transformations consist of node operations (e.g., insertions, deletions, inversions, and relabeling) as well as deletion of entire subtrees and paths. Insertion and deletion operations follow the semantics of the tree edit distance [5], while node inversions switch the position between a node and its parent; relabeling changes the node's label.

Error extent was defined as the percentage of tree nodes which were affected by the set of structural modifications. We considered as affected such nodes receiving modifications (e.g., a rename) and all its descendants. We classify the fuzzy copies generated from each data set according to the error extent used: we have *low* (10%), *moderate* (30%), and *dirty* (50%) error datasets. IDF is used as weighting scheme and WJS as set similarity function. All tests were performed on an Intel Xeon Quad Core 3350 2,66 GHz, about 2.5 GB of main memory. Finally, we conduct our experiments in the context of an XML DBMS platform called *XML Transaction Coordinator* (XTC) [4].

4.1 Accuracy Results

In our first experiment, we evaluated and compared the accuracy of similarity functions for ordered and unordered XML trees. For ordered trees, we employ *epq-grams* [6], an extension of the concept of *pq-grams* [7]. For unordered trees, we can exploit the fact that PCIs are already used to represent similar paths. For this, we simply use the PCIs corresponding to the set of paths of a tree to generate its profile: PCIs of a tree appearing in PC_t are appended to each *q-gram* generated from the corresponding text node and the remaining PCIs are used to directly represent structural tokens. We did not apply node-swapping operations when generating the dirty datasets; our comparison between similarity functions for unordered and ordered trees is fair. To evaluate accuracy, we used our join algorithms as selection queries, i.e., as the special case where one of the join partners has only one entry. We proceeded as follows. Each dataset was generated by first randomly selecting 500 subtrees from the original dataset and then generating 4500 duplicates from them (i.e., 9 fuzzy copies per subtree, total of 5000 trees). As the query workload, we randomly selected 100 subtrees from the generated dataset. For each queried input subtree T, the trees T_R in the result are ranked according to their calculated similarity with T; the relevant trees are those generated from the same source tree as T.



(a) *SwissProt*, varying error extent (b) *PSD*, varying error extent

Fig. 4: MAP values for different similarity functions on differing datasets

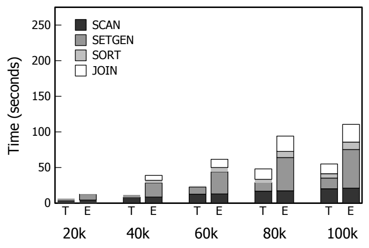
We report the *non-interpolated Average Precision* (AP), which is given by $AP = \frac{1}{\#relevanttrees} \times \sum_{r=1}^{n} [P(r) \times rel(r)]$, where r is the rank, n the number of subtrees returned. $P(r)$ is the number of *relevant* subtrees ranked before r, divided by the total number of subtrees ranked before r, and $rel(r)$ is 1, if the subtree at rank r is relevant and 0 otherwise. This measure emphasizes the situation, where more relevant documents are returned earlier. We report the mean of the AP over the query workload (MAP).

Figure 4 shows the results. Our first observation is that both similarity functions obtain near perfect results on low-error datasets. This means that duplicates are properly separated from non-duplicates and positioned on top of the ranked list. Even on dirty datasets, the MAP values are above 0.7 on *SwissProt* and 0.9 on *PSD*. In this connection, we observe that the results on *SwissProt* degrade more than those of *PSD* as the error extent increases. The explanation for this behavior lies on the flip side of structural heterogeneity: while providing good identifying information, structural heterogeneity severely complicates the selection of textual information and, thus, the set PC_t is more likely to contain spurious PCIs, especially on dirty datasets. Indeed, a closer examination on the dirty dataset of *SwissProt* revealed that PC_t contained, in fact, several unrelated paths. On the other hand, the results are quite stable on *PSD*, i.e., MAP values do not vary too much on a dataset and no similarity function experienced drastic drop in accuracy along differing datasets. Finally, PCI has overall better accuracy than EPQ (the only exception is on the dirty dataset of *PSD*).

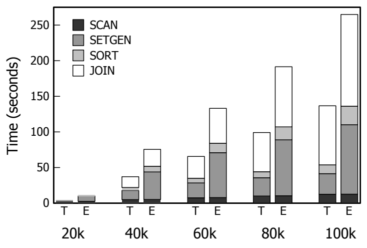
4.2 Runtime Performance and Scalability Results

In this experiment, we report the runtime results for fetching the input trees (SCAN), Profile Generation and Weighting steps (collectively reported as SETGEN), set collection sorting (SORT), and set similarity join (JOIN). Note that PCI and EPQ are abbreviated by P and E, respectively. We generated datasets varying from 20k to 100k, in steps of 20k. Finally, we fixed the threshold at 0.75.

The results are shown in Fig. 5. On both datasets, SCAN, SETGEN, and SORT perfectly scale with the input size. Especially for SCAN, this fact indicates that we achieved seamless integration of similarity operators with regular XQuery processing operators of XTC. SCAN is about 80% faster on *PSD* (Fig.



(a) SwissProt, 20–100k trees



(b) PSD, 20–100k trees

Fig. 5: TSJ execution steps on an increasing number of trees

5(b)) as compared to *SwissProt* (Fig. 5(a)), because characteristics of the *PSD* dataset lead to better compression rates of the storage representation. As a result, fewer disk blocks need to be read during the tree scan operation. On the other hand, SETGEN is about 2x slower on *PSD* as compared to *SwissProt* for both similarity functions. The text data of *PSD* defined by the path queries is larger than those of *SwissProt*, which results in larger sets and, in turn, higher workload for sorting and weighting operations. SETGEN is more than 3x faster on PCI as compared to EPQ. Because paths are provided for free by the path-oriented storage model, PCI-based profile generation simply consists of accessing the PCR-PCI table and splitting strings into sets of *q-grams*. On both datasets and for both similarity functions, SORT consumes only a small fraction of the overall processing time. In comparison to the other TSJ components, JOIN takes only up to 20% of the overall processing time on *SwissProt*, whereas it takes up to 60% on *PSD*; on the other hand, JOIN exhibits worst scalability.

5 Related Work

There is an extensive research literature on XML similarity, in which a large part is based on the concept of *tree edit distance* [5]. Despite its popularity, the tree edit distance is computationally expensive — worst case complexity of $O(n^3)$, where n is n is the number of nodes [8] — and, therefore, impractical for use in datasets with pontentially large trees. Guha et al. [9] proposed a framework where expensive distance computations are limited by using filters and a pivot-based approach to map trees into a vector space. Augsten et al. presented the concept of *pq-grams* to efficiently approximate the tree edit distance on ordered [7] and unordered trees [10]. Neither Guha et al. [9] nor Augsten et al. [7, 10] considers textual similarity. Weis and Naumann [11] proposed a framework for XML fuzzy duplicate identification. As in our approach, users can specify the textual information using selection queries. However, the framework only supports structured queries, which prevents its use on heterogeneous datasets.

Dalamagas et al. [12] exploited structural summaries to cluster XML documents by structure. Joshi et al. [13] employed the bag-of-tree-paths model, which represents tree structures by a set of paths. Our aim is completely different from these two previous approaches. We do not cluster XML documents directly; rather, we cluster paths to derive compact structural representations that can be, afterwards, combined with textual representations to calculate the overall tree similarity. Finally, this paper builds upon our own previous work on similarity for ordered trees [6], unordered trees [3], and set similarity joins [2].

6 Conclusion

We presented the design and evaluation of an approach to XML similarity joins. Focusing on token-based similarity functions, we decomposed the original problem into three components suitable for pipelined evaluation, namely, *profile generation*, *weighting*, and *set similarity*. XML tree profiles comprise tokens that capture structural information only and tokens representing structure together with user-defined pieces of textual information. In this context, we identified the need for similarity-based selection of text nodes to evaluate similarity joins on heterogeneous XML datasets — a so far neglected aspect — and proposed an approach that implicitly embeds similarity selection into join processing. We experimentally demonstrated that our approach can be applied flexibly to assess the similarity of ordered and unordered trees, deal with large datasets, and deliver accurate results even when the data quality decreases.

References

1. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. TKDE **19**(1) (2007) 1–16
2. Ribeiro, L.A., Härder, T.: Generalizing prefix filtering to improve set similarity joins. Information Systems **36**(1) (2011) 62–78
3. Ribeiro, L.A., Härder, T., Pimenta, F.S.: A cluster-based approach to xml similarity joins. In: IDEAS. (2009) 182–193
4. Mathis, C.: Storing, Indexing, and Querying XML Documents in Native XML Database Systems. PhD thesis, Technische Universität Kaiserslautern (2009)
5. Tai, K.C.: The tree-to-tree correction problem. Journal of the ACM **26**(3) (1979) 422–433
6. Ribeiro, L.A., Härder, T.: Evaluating performance and quality of xml-based similarity joins. In: ADBIS. (2008) 246–261
7. Augsten, N., Böhlen, M.H., Gamper, J.: The *pq*-gram distance between ordered labeled trees. TODS **35**(1) (2010)
8. Demaine, E.D., Mozes, S., Rossman, B., Weimann, O.: An optimal decomposition algorithm for tree edit distance. In: ICALP. (2007) 146–157
9. Guha, S., Jagadish, H.V., Koudas, N., Srivastava, D., Yu, T.: Integrating xml data sources using approximate joins. TODS **31**(1) (2006) 161–207
10. Augsten, N., Böhlen, M.H., Dyreson, C.E., Gamper, J.: Approximate joins for data-centric xml. In: ICDE. (2008) 814–823
11. Weis, M., Naumann, F.: Dogmatix tracks down duplicates in xml. In: SIGMOD. (2005) 431–442
12. Dalamagas, T., Cheng, T., Winkel, K.J., Sellis, T.K.: A methodology for clustering xml documents by structure. Information Systems **31**(3) (2006) 187–228
13. Joshi, S., Agrawal, N., Krishnapuram, R., Negi, S.: A bag of paths model for measuring structural similarity in web documents. In: SIGKDD. (2003) 577–582