Leveraging the Storage Layer to Support XML Similarity Joins in XDBMSs

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. XML is widely applied to describe semi-structured data commonly generated and used by modern information systems. XML database management systems (XDBMSs) are thus essential platforms in this context. Most XDBMS architectures proposed so far aim at reproducing functionalities found in relational systems. As such, these architectures inherit the same deficiency of traditional systems in dealing with less-structured data. What is badly needed is efficient support of common database operations under the similarity matching paradigm. In this paper, we present an engineering approach to incorporating similarity joins into XDBMSs, which exploits XDBMS components — the storage layer in particular — to design efficient algorithms. We experimentally confirm the accuracy, performance, and scalability of our approach.

1 Introduction

Emerging application scenarios increasingly require seamless management of data ranging from unstructured to structured format. An example of such scenarios is that of "schema-later" settings: data is initially loaded into repositories without prior schema definition or integration and queried through keyword search; structural information is added as more insight about the data is gleaned thereby enabling more complex queries.

XML is a core technology in the above context as it allows describing and querying heterogeneous data that exhibits varying degree of structure using a single data model. As such, XDBMSs are natural candidates to serve applications with features such as efficient management and controlled access of data [1, 2]. Unfortunately, while successful at reproducing most of the functionalities commonly found in traditional relational systems, most XDBMSs proposed so far fall short in effectively dealing with non-schematic XML databases. The main reason is the strict focus — inherited from relational systems — on structured, Boolean queries based on exact matching. However, queries on less-structured data invariably require matching the data based on some notion of similarity.

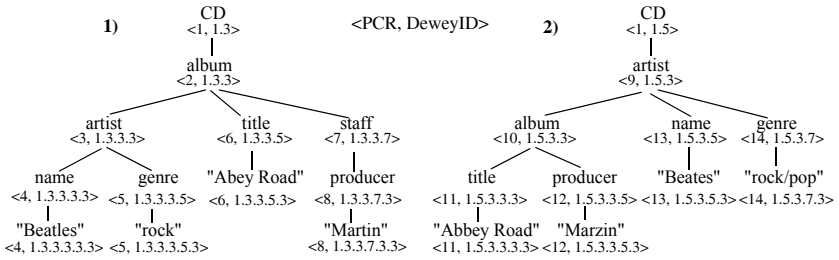


Fig. 1: Heterogeneous XML data

For relational data, such similarity matching is often needed on text fields owing to misspellings and naming variations. For XML — commonly modeled as a labeled tree —, similarity matching 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 (disregard the values associated with each XML node for the moment). The integration of these databases may require identifying multiple representations of real-world entities; this task is called *fuzzy duplicate identification*, among many other terms used [3]. Indeed, 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 (e.g., "Beatles" and "Beates").

In this paper, we investigate the integration of similarity joins into XDBMSs. Most of previous work addressing the integration of similarity (or relevance) concepts into XML queries has viewed the similarity computation operation as "black-boxes", i.e., implementation-dependent operations that are invoked and manipulated within a framework composed by host language and algebraic constructs [4]. In contrast, we tackle here the problem of seamlessly and efficiently evaluating such operations in an XDBMS context. We exploit several XDBMS-specific features; in particular, we push down a large part of the structural similarity evaluation close to the storage layer. To the best of our knowledge, our work is the first to leverage the storage model of XDBMSs for XML similarity join processing. Our engineering approach only requires simple additional data structures, which can be easily built and maintained. The resulting similarity operator can be plugged to logical algebras for full-text search [4] as well as composed with regular queries. Finally, we show through experiments that the our proposal delivers performance and scalability without compromising accuracy thereby providing a suitable platform for managing semi-structured data.

The rest of this paper is organized as follows. Sect. 2 gives the preliminaries and Sect. 3 formally defines the similarity join operator. Sect. 4 overviews the underlying XDBMS architecture. Similarity algorithms are presented in Sect. 5, building and maintenance of auxiliary structures are covered in Sect. 6, and the composition of query plans is described in Sect. 7. Experiments and related work are discussed in Sect. 8 and Sect. 9, respectively. Sect. 10 concludes this paper.

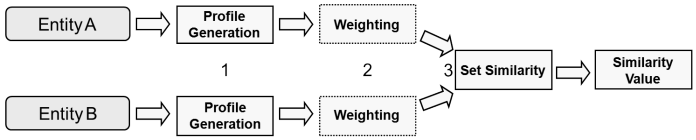


Fig. 2: Evaluation of token-based similarity functions

2 Preliminaries

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 a set representation the *profile* of an entity, the elements of the profile are called *tokens*, and, optionally, a *weighting scheme* can be used to associate weights to tokens. 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 textual and structural information are presented in Sect. 5.

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 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∘1, b∘1, b∘2} (the symbol ∘ denotes concatenation).

Weighting Schemes 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. The Level-based Weighting Scheme (*LWS*) [5] 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 [6].

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

2.2 XML Path Clustering

We now briefly review our approach based on path clustering, which provides the basis for combining and calculating structural and textual similarities and generating compact profiles. For a detailed discussion, please see [5].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 is in pc^i.

Prior to clustering, all path classes of a database have to be 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 [2]. Each node in a PS represents a (partial) path class and is identified by a *Path Class Reference* (PCR). Note that we can establish an association between PCR and PCI values on leaf nodes. Fig. 3 depicts the PS of the sample database shown in Fig. 1, where the association between PCR and PCI values is explicitly represented in the so-called *PCR-PCI table*. 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

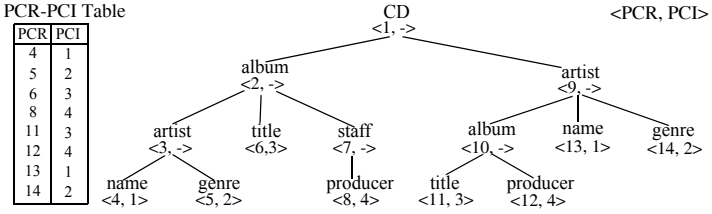


Fig. 3: Path synopsis annotated with PCR and PCI values

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 *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 not less than a given threshold. Note that all elements of the definition below can be flexiby combined to yield different similarity functions.

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 $sf(T_1, T_2) = \tau\prime \geq \tau$.*

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 the host XDBMS 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

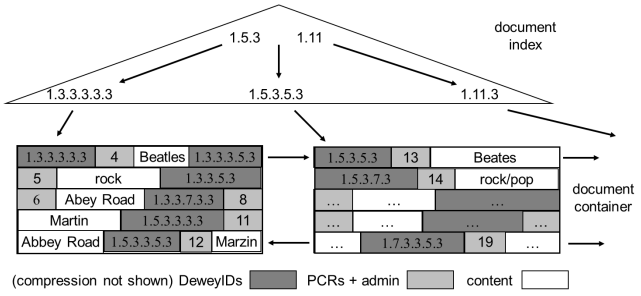


Fig. 4: Stored document in path-oriented format

implemented and evaluated in a pipelined fashion. Profile Generation is the most important step and will described in detail in Sect. 5. For Weighting, we maintain the frequency of all tokens in the database in a simple memory-resident *token-frequency table*. Set Similarity is implemented by the set similarity join algorithm based on inverted lists presented in [6]. 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)**.

4 The XML Transaction Coordinator

We use an XDBMS platform called *XML Transaction Coordinator* (XTC) [2] as a vehicle to deliver a proof of concept and performance of our approach. XTC employs a node identification scheme called *DeweyIDs*, which allows processing (part of) queries without touching physical storage. A DeweyID encodes the path from the documents' root to the node as well as sibling order and is represented by a sequence of integers called *divisions*. Trees identified by DeweyIDs are shown in Fig. 1. DeweyIDs capture a large share of the structural information including all structural relationships between two nodes and node's ancestor ID list.

DeweyIDs and PS structure described in Sect. 2.2 are complementary: by associating the DeweyID of a node with the corresponding node in the PS, we are able to derive not only the label of this node, but also its complete path information. Consider again Fig. 1. Note that element nodes are associated with the respective PCR in the PS; text nodes are associated with the PCR of its owing element. The PCR value for the text node with value "Beatles" and DeweyID 1.3.3.3.3.3 is 4. Using the DeweyID in conjunction with the PS structure illustrated in Fig. 3, we can now reconstruct the path from the root to this node, therefore obtaining $\langle$(CD,1.3),(album,1.3.3),(artist,1.3.3.3), (name,1.3.3.3.3)$\rangle$.

The ability of reconstructing paths instigates the design of a space-economic storage model: only text node values of XML documents need to be stored together with the corresponding DeweyID and PCR. The structural part is then virtualized and can be reconstructed whenever needed. This model is referred

to as the *path-oriented storage model*—contrast it with the node-oriented storage model, where all nodes, structural and textual, are stored on disk. Path-oriented storage can achieve substantial space savings without degrading the performance of operations such as tree reconstruction and navigation [2]. Fig. 4 zooms in on the path-oriented format. The *document index*, a B-tree with key/pointer pairs $\langle DeweyID, PagePtr \rangle$, indexes the first text node in each page of a set of doubly chained pages. Text nodes of variable length are represented by $\langle DeweyID, PCR, value \rangle$ and stored in document order.

5 Profile Generation

We now present algorithms for profile generation of ordered and unordered XML trees. As general strategy, text data appearing under a path in PC_t is converted to a set of *q-grams* and appended to structural tokens.

5.1 Ordered Trees

For ordered trees, we employ *epq-grams* [7], an extension of the concept of *pq-grams* [8]. Informally, all subtrees of a specific shape are called *pq-grams* of the corresponding tree. A *pq-gram* consists of an *anchor node* prepended of $p-1$ ancestors, called *stem*, and q children, called *base*. To be able to obtain a set of *pq-grams* from any tree shape, an expanded tree is (conceptually) constructed from the original tree by extending it with *dummy nodes*; please see [8] for details. The *pq-gram profile* of a tree is generated by collecting all *pq-grams* of the respective expanded tree. In this context, *epq-grams* are generated by carefully representing text data in the expanded tree. Text nodes are treated differently depending on which node is currently selected as anchor: each character of a text node data is used as *character node* when its parent is the anchor node, and *q-gram nodes* are used when the text node itself is the anchor [7].

The difference between the algorithms for generation of *epq-gram* and *pq-gram* profiles lies, of course, in the way that text nodes are handled. Thus, we only discuss this aspect now. Following the same notation in [8], the stem and the base are represented by two shift registers: *anc* of size p and *sib* of size q. The *shift* operation is used to remove the head of the queue and insert a new element at the tail of the queue, i.e, $shift((a,b,c),x) = (b,c,x)$. We further define the operation *tail*, which substitutes the element at the tail of the queue by a new element, i.e., $tail((a,b,c),x) = (a,b,x)$. A snippet of the algorithm for the generation of *epq-gram* profiles is listed in Alg. 1 (the remaining parts of the algorithm generate regular *pq-grams* [8]). When iterating over the children of a node u, we check if the current child is a text node (line 14) and if its corresponding PCR is in PC_t (line 15). If both conditions hold, we generate the corresponding *pq-gram* tokens (lines 16–22). *epq-gram* tokens are formed by stems having either u as anchor (represented by *anc*) or a *q-gram* token (represented by the register *anc-p*). In the loop at line 18, the algorithm iterates over the set of tokens returned by the *qgram* function and composes *epq-gram* tokens by concatenating *anc* with *sib* and *anc-p* with a concatenation of q dummy nodes (represented by *qdummy*).

Algorithm 1: Algorithm for the generation of *epq-gram* tokens

Input: A tree T, positive integers p and q, the set PC_t
Output: The *epq-gram* profile $\mathcal{P}$ of T

```
1  ...
13 foreach for each child c (from left to right) from u do
14     if c is a text node then
15         if PCR-PCI(u.pcr) ∈ PC_t then
16             qdummy ← concatenation of q dummy nodes
17             anc-p ← shift(anc, *)
18             foreach tok ∈ qgram(c, q) do
19                 sib ← shift(sib, tok)
20                 P ← P ∪ (anc ∘ sib)
21                 anc-p ← tail(anc-p, tok)
22                 P ← P ∪ (anc-p ∘ qdummy)
23         continue
24     ...
25 return P
```

5.2 Unordered Trees

For unordered trees, we can exploit the fact that PCIs are already used to represent similar paths. In other words, similarity matching between paths is reduced to a simple equality comparison between their corresponding PCIs, because the actual path comparison has already been performed during the clustering process on the PS structure. Hence, 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. The algorithm for the generation of PCI-based profiles is shown in Alg. 2. The simplicity of the algorithm reflects the convenience of the path-oriented storage model for our PCI-based tokenization method. First, only leaf nodes are needed to derive a tree representation. Hence, the reconstruction of inner nodes is obviated. Further, PCIs are obtained by a simple lookup at the PCR-PCI table (line 2); the corresponding tokens are obtained directly from the PCI value (line 6) or by the concatenation $pci \circ tok$ (line 4). As a result, profile generation can be implemented by a lightweight operator that imposes very little overhead to the overall similarity processing.

6 Auxiliary Structures: Building and Maintenance

We have to maintain two auxiliary data structures: the PCR-PCI table and the token-frequency table. All of them are kept memory-resident during similarity join evaluation and are incrementally updated as the database state changes. The PCR-PCI table is built at the end of the path clustering process described

Algorithm 2: Algorithm for the generation of PCI-based tokens

Input: A set of text nodes and null nodes N from a tree T, an positive integer q, the set PC_t, the PCR-PCI table
Output: The PCI-based profile $\mathcal{P}$ of T

```
1 foreach u ∈ N do
2     pci ← PCR-PCI(u.pcr)
3     if u ∈ PC_t then
4         foreach tok ∈ qgram(u, q) do
5             P ← P ∪ (pci ∘ tok)   // insert new pci-qgram into the profile
6     else
7         P ← P ∪ pci   // insert new pci into the profile
8 return P
```

in Sect. 2.2 and has a reasonably small memory footprint: it requires 4 bytes per entry (two bytes each for PCR and PCI) where the number of entries is given by the number of distinct paths in the dataset. Modifications on the PS, i.e., path class insertions or deletions, have to be propagated to the PCR-PCI table. This task is trivial: we only need to remove or insert a table entry. When a new path class is inserted, we identify the cluster most similar to it in the same way as for path queries (see Sect 2.2) and update the PCR-PCI table.

We build the token-frequency table by performing a single sweep over the database, typically right after the clustering process. As distinct PC_t sets yield different token sets, we need a mechanism to provide token-frequency information for any PC_t set defined by the user. To this end, we adopt the simple solution of generating all possible tokens, i.e., we generate tokens for $PC = PC_t$ and $PC_t = \oslash$. For *PCI*-based profiles, we use a slightly adapted version of Algorithm 2, where lines 4–5 and line 7 are executed for all input nodes. For *epq*-gram profiles, we execute both the Algorithm 1 and the original *pq*-gram algorithm and take the duplicate-free union of the resulting profiles. Note that we have to build a token-frequency table for each tokenization method as well as for different parameterizations thereof (e.g., q-gram size). Despite the large number of tokens, the frequency table is still small enough to typically fit into main memory. For example, using real-world XML databases (see Sect. 8), we have less than 40K distinct tokens per database. The frequency table requires 8 bytes per entry (four bytes each for the hashed token and its frequency); thus, only around 312KB are sufficient to keep the frequencies of all tokens memory-resident.

For *PCI*-based tokens, updating the token-frequency table after data changes is easy. In case of deleting structure nodes, content nodes, or both, we only need to generate the tokens for the deleted data to probe the token-frequency table and decrease the corresponding frequency by one—tokens with frequency zero are removed from the table; in case of insertions, we generate the tokens for the new data and increment their frequency accordingly or add an entry in the token-frequency table for new tokens. For *epq*-gram tokens, incremental updates are more complicated due to the underlying sibling ordering which imposes more

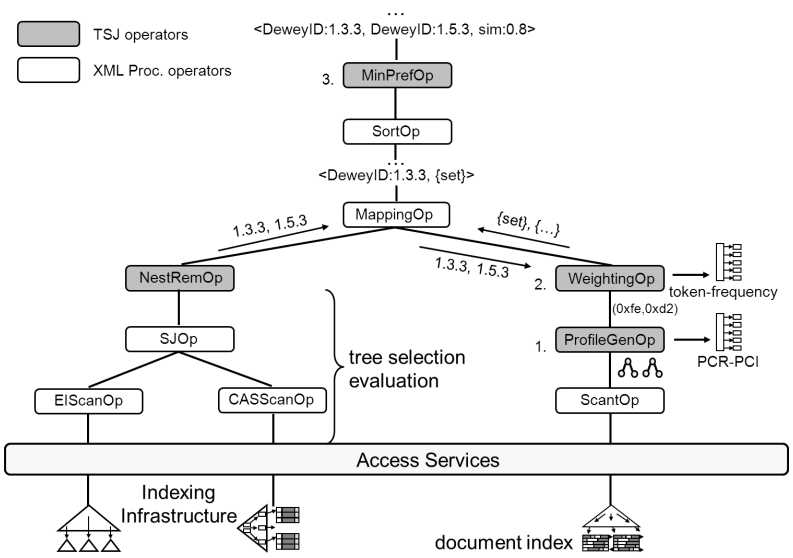


Fig. 5: TSJ query evaluation plan

data dependency on the token generation process. Currently, we apply the profile generation algorithm on the whole tree to update the token-frequency table.

7 TSJ as a Query Evaluation Plan

We now put all TSJ components together to compose a complete similarity join operator. The main design objectives are *seamless integration* of the TSJ operator into XDBMS's architecture and *performance*; the former is achieved by encapsulating TSJ components and assembling them into a QEP, while the latter is obtained by enabling pipelining as much as possible. Figure 5 illustrates a TSJ plan. Physical operators are represented by rounded rectangles; operators specific for similarity join processing are highlighted with a shade of gray. Parts of the operator tree that closely correspond to the steps in the course of token-based similarity evaluation (see Fig. 2) are identified by the respective numbers.

Following the dataflow order, the left-hand side branch at the lower end of the QEP executes the tree selection expression exploiting available indexes to obtain an initial list of root-node DeweyIDs. The *NestRemOp* operator simply removes nested DeweyID sequences by retaining only the topmost DeweyID. Completing the tree access pre-step, the list of DeweyIDs is streamed along the path to the ScanOp operator, which fetches a tree at a time using the document index. The following two components upwards are straightforward implementations of the steps 2 (*Profile Generation*) and 3 (*Weighting*): trees represented by sets of nodes are converted into profiles by *ProfileGenOp* using the PCR-PCI table and, afterwards, the profiles are converted to (weighted) sorted sets using the token-frequency table. Root DeweyIDs and the corresponding sorted set are combined

by *MappingOp* and sorted in increasing order of the set size by *SortOp*. Finally, the *Set Similarity* step is performed by *MinPrefOp* and scored pairs of DeweyIDs are delivered to the TSJ consumer. TSJ can be used as stand-alone operator and as part of more complex XML queries. For instance, we can simply plug-in a sort operator on top of the QEP to deliver the resulting DeweyIDs in document order.

8 Experiments

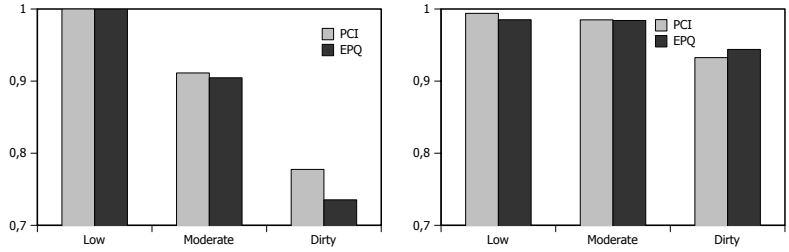
We used two real-world XML databases on protein sequences, namely *SwissProt* (http://us.expasy.org/sprot/) and *PSD* (http://pir.georgetown.edu/). We deleted the root node of each XML dataset to obtain sets of XML documents. The resulting documents are structurally very heterogeneous. On average, *SwissProt* 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. 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 [9], 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.

8.1 Accuracy Results

We evaluated and compared the accuracy of the similarity functions based on PCIs and *epq-grams* (EPQ). Note that we did not apply node-swapping operations when generating the dirty datasets; hence, our comparison between similarity functions for unordered and ordered trees is fair. We used our join algorithms as selection queries, i.e., as the special case where one of the join partners has only one entry. Each dataset was generated by first randomly selecting 500 subtrees from the original dataset and then generating 9 fuzzy copies per subtree (total of 5k 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



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

Fig. 6: MAP values for different similarity functions on differing datasets

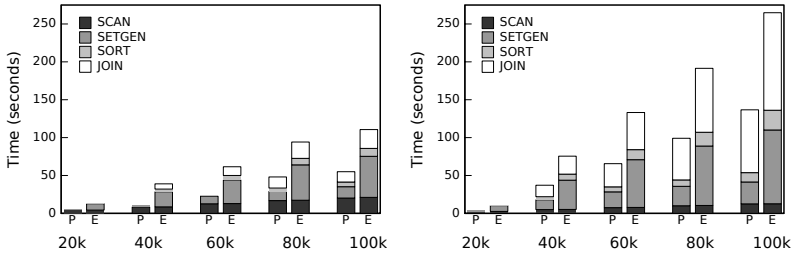
the result are ranked according to their calculated similarity with T; relevant trees are those generated from the same source tree as T.

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). In addition, we experimented with several other metrics such as the *F1* measure and obtained similar results.

Figure 6 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*).

8.2 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 SET-GEN), 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.



(a) SwissProt, 20–100k trees (b) PSD, 20–100k

Fig. 7: TSJ execution steps on an increasing number of trees

The results are shown in Fig. 7. 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. SCAN is about 80% faster on *PSD* (Fig. 7(b)) as compared to *SwissProt* (Fig. 7(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 the worst scalability.

9 Related Work

Guha et al. [9] presented an approach to XML similarity joins based on tree edit distance, while that of Augsten et al. [8] is based on *pq-grams*. None of them considers textual similarity. XML retrieval based on similarity matching and ranking have been intensively studied over the last few years; representative techniques are based on query restriction relaxation [10] and keyword search [11]. We are not aware of any work that leverages the storage layer of the underlying system to support similarity evaluation.

TIMBER [1] is one of the first XDBMSs described in the literature. Al-Khalifa et al. [4] integrated full-text search in TIMBER by extending the existing data model and algebra to support scores and developing new physical operators. TopX [12] is an example of a system designed from scratch for supporting ranked XML retrieval. Similarity joins are not considered in TopX. Further, Oracle

RDBMS supports the Binary XML storage format, which enables efficient Xpath pattern matching [13]. Similarity matching is not considered, however.

This paper complements our own previous work on similarity for ordered trees [7], unordered trees [5], and set similarity joins [6] by putting them together into an XDBMS and exploiting specific components of the environment. Our initial version of TSJ within XTC followed an implementation based on relational operators [7]; the implementation presented in this paper is radically different and outperforms this previous version by orders of magnitude.

10 Conclusion

In this paper, we presented an engineering approach to integrate XML similarity joins into XDBMSs exploiting several components of the existing architecture. In particular, the so-called path-oriented storage model was found to be a perfect match to our similarity functions, which enabled the design of inexpensive algorithms. These algorithms are implemented as physical operators that can be flexibly assembled into query evaluation plans. Effectiveness, performance, and scalability of our solution were successfully validated through experiments.

References

1. Jagadish, H.V., et al.: Timber: A native xml database. VLDB J. **11**(4) (2002) 274–291
2. Mathis, C.: Storing, Indexing, and Querying XML Documents in Native XML Database Systems. PhD thesis, Technische Universität Kaiserslautern (2009)
3. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. TKDE **19**(1) (2007) 1–16
4. Al-Khalifa, S., Yu, C., Jagadish, H.V.: Querying structured text in an xml database. In: SIGMOD. (2003) 4–15
5. Ribeiro, L.A., Härder, T., Pimenta, F.S.: A cluster-based approach to xml similarity joins. In: IDEAS. (2009) 182–193
6. Ribeiro, L.A., Härder, T.: Generalizing prefix filtering to improve set similarity joins. Information Systems **36**(1) (2011) 62–78
7. Ribeiro, L.A., Härder, T.: Evaluating performance and quality of xml-based similarity joins. In: ADBIS. (2008) 246–261
8. Augsten, N., Böhlen, M.H., Gamper, J.: The *pq*-gram distance between ordered labeled trees. TODS **35**(1) (2010)
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. Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D., Toman, D.: Structure and content scoring for xml. In: VLDB. (2005) 361–372
11. Chen, Y., Wang, W., Liu, Z.: Keyword-based search and exploration on databases. In: ICDE. (2011) 1380–1383
12. Theobald, M., et al.: Topx: efficient and versatile top- k query processing for semistructured data. VLDB J. **17**(1) (2008) 81–115
13. Zhang, N., et al.: Binary xml storage and query processing in oracle 11g. PVLDB **2**(2) (2009) 1354–1365