Evaluating Performance and Quality of XML-Based Similarity Joins

Leonardo Ribeiro and Theo Härder

AG DBIS, Department of Computer Science,
University of Kaiserslautern, Germany
{ribeiro,haerder}@informatik.uni-kl.de

Abstract. A similarity join correlating fragments in XML documents, which are similar in structure and content, can be used as the core algorithm to support data cleaning and data integration tasks. For this reason, built-in support for such an operator in an XML database management system (XDBMS) is very attractive. However, similarity assessment is especially difficult on XML datasets, because structure, besides textual information, may embody variations in XML documents representing the same real-world entity. Moreover, the similarity computation is considerably more expensive for tree-structured objects and should, therefore, be a prime optimization candidate. In this paper, we explore and optimize tree-based similarity joins and analyze their performance and accuracy when embedded in native XDBMSs.

1 Introduction

Data cleaning deals with the identification and correction of data inconsistencies. Frequently, due to such inconsistencies (e.g., mis-spellings), multiple representations of real-world objects appear in data collections. Such redundancy may lead to wrong results, confuse consistency maintenance, and, when integrated from various data sources, may artificially inflate data files. Therefore, detection of duplicates by correlation of (string) attribute values is a long-term research goal in the relational world [5,6,4,2,1], often denoted as the *fuzzy duplicate problem.*

As a classical solution, relational DBMSs have correlated records by using *similarity joins.* A similarity join pairs tuples from two relations whose specified attribute values are similar. These values can be composed by a single column or by the concatenation of column sequences (e.g., *R[Name]* and *R[Name, Address]*). Using a *similarity function*, a pair of tuples is qualified if a similarity value greater than a given threshold is returned.

Over the recent years, XML is increasingly used as standard for information representation, in addition to provide a common syntax for data exchange. In this context, similarity operators must deal with tree-structured documents instead of table-structured objects. However, extending such operations to XML brings a new quality dimension to the correlation problem. Because similar or even the same information could be embodied by quite different structures or fragments in XML documents, textual techniques developed for relational data are not

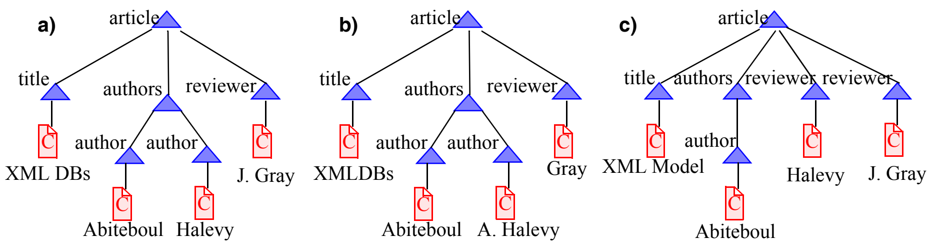


Fig. 1. Sample XML document fragments

sufficient, as disclosed by the example in Fig. 1. Consider a situation where tree *a)* has to be correlated to tree *b)*. Although they are obviously identical (for human observers), they would not be classified as equal because of variations in the XML *content part*. However, the use of an appropriate textual similarity predicate would easily classify them as matching candidates. On the other hand, although *c)* refers to another article, the comparison of *a)* with *c)* using the same similarity predicate would probably lead to an erroneous classification, i.e., to a match, when only textual similarity is considered. In this case, the information needed to lower the overall similarity is conveyed in the XML *structure part*. Due to the increased modeling flexibility of XML, e.g., by optional elements and attributes, even data sources sharing a same DTD may not have an identical tree structure. Therefore, accurate and efficient DBMS methods to correlate XML data need to cope with additional complexities induced by the XML structure.

Our Contribution. We propose built-in support for similarity joins in an XDBMS query engine and design a foundational framework for approximation operators based on three main conceptual components: system embedding, candidate pair generation, and quality measures. System embedding means subtree access using index-based location of qualified XML fragments. We complement this component with the support of various types of predicates to select specific parts of a subtree for similarity evaluation. Using techniques from the relational world such as signature schemes and equi-joins, we facilitate candidate pair generation. For quality measures, we concentrate on set-overlap-based similarity functions. In this setting, we introduce a novel mechanism, *extended pq-grams*, to derive tokens combining structure and content of XML tree structures. For this new concept, we propose three versions of token generation which jointly use textual and structural information. Furthermore, we explore a tokenization scheme called *path-gram*. Our solution has important advantages over previous work on XML data: it unifies string- and tree-based techniques in a single processing and scoring model thereby providing efficiency and improving the overall matching quality; the resulting operation can be combined with regular XML queries (e.g., by delivering the resulting nodes in document order) to compose processing logic which enables more complex data cleaning solutions. Finally, we give detailed quantitative results by conducting empirical experiments and performance measurements using our prototype XDBMS called XTC [10].

2 Related Word

Chaudhuri et al. [4] introduced the idea of extending the set of physical operators to provide support for similarity joins inside database engines. A core component of this approach uses signature schemes, which is used to avoid unnecessary comparisons[2]. Alternatively, similarity joins can be specified by using languages such as SQL [5,6,1]. Such a specification offers high flexibility and enables system-controlled optimization as key advantages. However, the related approaches have some serious performance limitations. First, they require considerable effort in a pre-processing stage where a number of auxiliary tables have to be constructed, e.g., tables to store tokens and cardinalities. Second, the set of candidates consists of all pair of tuples that share at least one token. Even by using well-known techniques such as stop-word removal, the set of candidates can potentially become very large and can be frequently populated by tuples that do not make it to the final result. A solution proposed for this problem uses sampling methods to one or both of the join partners [5], and thus are compromising the *exactness* of the result, i.e., they may miss some valid results.

A very large body of work is available on textual similarity [4,13] for which we only mention the well-known concept of *edit distance* [11]. More important for our work is the *tree edit distance* which—using tree edit operations such as node insertion, node deletion, and node renaming—is defined as the cost-minimal operation sequence transforming a tree into the one to be compared [15]. Unfortunately, all algorithmic results have more than $O(n^2)$ runtime complexity and are, therefore, impractical for XDBMS use with potentially large trees.

Close to our idea, Guha et al. [7] present a framework for XML similarity joins based on tree edit distance. To improve scalability, they optimized and limited distance computations by using lower and upper bounds for the tree edit distance as filters and a pivot-based approach for partitioning the metric space. However, these computations are still in $O(n^2)$ and heavily depend on a good choice of parameters (reference set). Avoiding this cost/scalability penalty and manual parameter choices, applying *pq*-grams [3] to derive an efficient tree-based distance approximation seems to be more interesting for XDBMS use. Moreover, the use of structural information enabled by a generalization of *q*-grams allows leveraging a large body of similarity join techniques addressing performance enhancements (e.g., *signature schemes* [2,4] and pipelined evaluation [4]) as well as versatile applications of similarity functions (e.g., set-overlap-based similarity methods [4,13]). Moreover, in contrast to [7], this framework easily deals with modifications of the underlying tree structures.

3 Concepts Used for Similarity Join Computation

An XML document is modeled as an ordered labeled tree. We distinguish between element nodes and text nodes, but not between element nodes and attribute nodes; each attribute is child of its owning element. Disregarding other node types such as Comment, we consider only data of string type.

3.1 Similarity Joins on XML Collections

A general tree similarity join takes as input two collections of XML documents (or document fragments) and outputs a sequence of all pairs of trees from the two collections that have similarity greater than a given threshold. The notion of similarity between trees is numerically assessed by a similarity function used as join predicate and applied on the specified node subsets of the respective trees.

Definition 1 (General Tree Similarity Join). *Let F_1 and F_2 be two forests of XML trees. Given two trees T_1 and T_2, we denote by $sim(T_1, T_2)$ a similarity function on node sets of T_1 and T_2, respectively. Finally, let γ be a constant threshold. A tree similarity join between F_1 and F_2 returns all pairs $(T_1, T_2) \in F_1 \times F_2$ such that $sim(T_1, T_2) \geq \gamma$.*

Note that the similarity function is applied to node sets instead to trees. When comparing trees, we need the flexibility to evaluate their similarity using node subsets that do not have containment relationships among them, e.g., node sets only consisting of text nodes. If structure matters, the *node labeling scheme* allows identifying containment relationships among a set of nodes.

3.2 Set-Overlap-Based Similarity Measures

Given two sets representing two objects, different ways to measure their overlap raise various notions of similarity (or dissimilarity). There are several proposals for such measures, among others the *Jaccard similarity*, binary cosine similarity, and the Hamming distance. We observed that the method used to map an object to a set also has influence on the notion of similarity, because it determines which properties of the object are under consideration by the similarity measure. For example, given an XML tree, we can produce sets representing its textual information or its structural information (in the approximation sense). Therefore, the overall set-overlap-based similarity calculation unfolds two operations that can be independently dealt with: *conversion* of objects to sets and, afterwards, *set-overlap* measurement. As an example, consider the well-known Jaccard similarity that, for two sets let r and s, is given by: $Jacc\,(r, s) = \left|\frac{r \cap s}{r \cup s}\right|$.

3.3 Signature Schemes

To avoid bulky similarity evaluation for each pair of sets, a *signature scheme* is commonly used. Given a collection of sets as input, a signature scheme produces a shorter representation of each set, called signature, that roughly maintains their pairwise similarity according to a given measure. An essential *correctness requirement* is that *false negatives* must not be produced [2]: for any two sets r, s, and their respective signatures $Sig(r)$, $Sig(s)$, we have $Sig(r) \cap Sig(s) \neq \varnothing$ whenever $sim(r, s) \geq \gamma$.

One previously proposed signature scheme is the prefix-filter [4], which is based on the following intuition: for two sets r and s of size c under a same *total order*, if $|r \cap s| \geq \gamma$, then subsets consisting of the first $c - \gamma + 1$ elements of

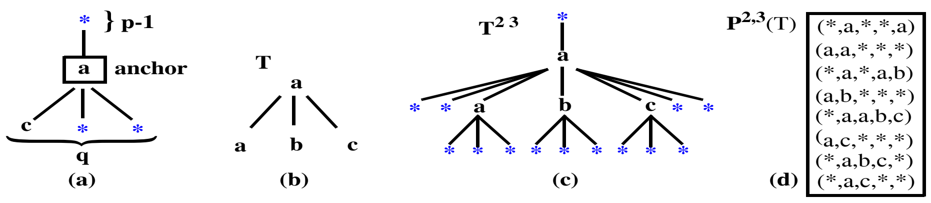


Fig. 2. Steps for the generation of *pq*-gram tokens

r and s should intersect. Minor variations of this basic idea are used to handle weighted sets and normalized similarity functions. Please, see [4] for details. The ordering of the sets is picked to keep the elements with smallest frequencies in the signature, i.e., the elements are ordered by increasing frequency values.

4 Mapping Trees to Sets by Token Generation

The technique of decomposing a string in substrings of length q, the so-called *q-grams*, is widely used in the approximate string matching area [11]. The main idea is to assess the "closeness" between two strings by using the overlap of their sets of q-grams [14]. Hence, q-grams provide a natural choice for their use in conjunction with set-overlap-based similarity measures. Moreover, this approach carries over to tree-structured data as well: trees can be split into subtrees of a same shape and the structural similarity between two trees can be calculated on basis of the number of common subtrees. Next, we review a generalization of q-grams for structural similarity assessment and then present approaches for combining structural and textual similarity into a single measure.

4.1 The Concept of *pq*-Grams

The concept of *pq*-grams was presented by Augsten et al. [3] to map a *tree structure* to a set of tokens. All subtrees of a specific shape are denoted *pq*-grams of the corresponding tree. This shape is defined by two positive integer values p and q: a *pq*-gram consists of an *anchor node* together with $p-1$ ancestors and q children, as visualized by a sample *pq*-gram in Fig. 2*a*. The concatenation of the node labels of a *pq*-gram forms a *pq*-gram token (*pq*-gram, for short). To be able to obtain a set of *pq*-grams from any tree shape, an expanded tree $T^{p,q}$ is (conceptually) constructed from the original T by inserting *null nodes* as follows: $p-1$ ancestors to the root node; $q-1$ children before the first and after the last child of each non-leaf node and q children to each leaf node. Fig. 2*b* and *c* show tree T and its expanded form $T^{2,3}$. A *pq-gram profile* is obtained by collecting all the *pq*-grams of an expanded tree resulting in a profile cardinality $|P^{p,q}(T)| = 2l + kq - 1$ for a tree with l leaf nodes and k non-leaf nodes. Fig. 2*d* shows the *pq*-gram profile of $T^{2,3}$.

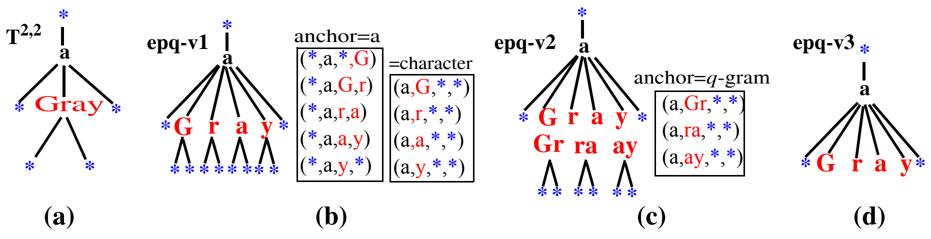


Fig. 3. Versions of extended *pq*-grams

The set of *pq*-grams can be easily calculated using a preorder traversal of the corresponding tree [3]. In our case, the input is a stream of structure nodes in *document order* provided by the underlying document access mechanism.

4.2 Adding Content Information

We now study ways to simultaneously deal with structural and textual similarity. The notion of *pq*-grams captures variations in the XML structure part, but entirely ignores the presence of text variations, i.e., deviations in XML text nodes. Hence, it falls short in identifying duplicates in datasets with poor textual data quality. If we disregard the structure using *q*-grams only, we may incur in the problem described in Sect. 1: unrelated content may be compared leading to wrong matching results. Hence, an intuitive approach is to produce tokens (grams) that jointly capture structural and textual properties of a tree.

To achieve the objective above, we propose an extension of the original definition of *pq*-grams. When the objects of interest are represented by strings, their sets of *q*-grams are used to enrich the respective *pq*-grams, leading to *extended pq-grams* (*epq*-grams for short). For *epq*-gram generation, we focus on the conceptual representation of strings in an expanded tree, denoted by $ET^{p,q}$. This approach allows us to use an almost identical algorithm to produce *epq*-grams from a stream of nodes, with minor variations to handle text nodes. Furthermore, the generated grams seamlessly reduce to normal *pq*-grams when the input stream only contains structural nodes. There are several conceivable ways to represent strings as nodes in an expanded tree. Next, we analyze three versions.

The first alternative consists of considering each character of a string as a *character node*. Hence, whenever a parent of a text node is selected as an anchor node, *q* character nodes are selected to form a new *epq*-gram version called *epq*-v1. Given a $T^{2,2}$ in Fig. 3*a*, the corresponding $ET^{2,2}$ for *epq-v1* together with the resulting *epq*-profile are shown in Fig 3*b*. Note that the *epq*-profile is separated into two subsets: *epq*-grams having *a* as anchor node, the other having character nodes as anchor node. When the node labels are concatenated, sequences of character nodes form *q*-grams, which are combined with structural information. Unfortunately, *epq*-v1 always forms *1*-grams when the character node is the anchor, which is independent of the choice of *q*. Note that for $q = 1$, different strings containing an identical (*multi-*) set of characters have the same *q*-gram profile and, hence, maximum similarity. For $q > 1$, more complex patterns must be present to get this (unwanted) effect [14].

To prevent from the potential drawback of *epq*-v1, we propose a hybrid approach, called *epq*-v2 (see Fig. 3*c*). Now, character nodes are used when the parent is the anchor node, and *q-gram nodes* when the text node itself is the anchor. As a result, all *epq*-grams with textual information have q-grams of the same size (*epq*-grams having a as anchor node are the same to those of *epq*-v1).

The previous versions may consume substantial space because of large profile sizes. This observation motivates the third approach, *epq-v3*, which is derived by using character nodes, but pruning their *q-null* children from the expanded tree (see Fig. 3*d*). Compared to the previous versions, this approach roughly produces only half of the *epq*-grams embodying text; therefore, textual similarity receives less weight. However, this property can be compensated by the fact that tokens containing text are likely to be less frequent than structure-only tokens. The rationale is that by using common notions of weights that are inversely proportional to frequency, e.g., IDF, we can balance the effect of the q-gram reduction. In Sect. 6, we empirically evaluate this conjecture.

Theorem 1 shows the relation between the resulting profile cardinality and the number of non-leaf nodes, empty nodes, and text nodes.

Theorem 1. *Let $p > 0$, $q > 0$, and T be a tree with e empty nodes, k non-leaf nodes and t text nodes. Assume that all text nodes have a fixed length of n. The size of the extended epq-gram profile (version 1) is: $|(EP_{v1})^{p,q}(T)| = kq + 2e + 2tn - 1$.*

Proof (Sketch). Theorem 1 can be shown by structural induction similarly to the strategy used in [3]. The deletion of leaves should be done in two stages: first deletion of empty nodes and then deletion of text nodes. Deleting a text node decreases the cardinality of the *pq*-gram profile by $2n$ if the text node has siblings, otherwise by $2n + q - 2$; deletion of an empty node decreases the cardinality by q if the node has no siblings, otherwise by 2.

Similarly, the profile cardinality can be derived for version 2 and 3 leading to $|(EP_{v2})^{p,q}(T)| = kq+2e+t(2n - q + 1)-1$ and $|(EP_{v3})^{p,q}(T)| = kq+2e+tn-1$.

Especially when applied to text nodes with long strings, the *epq*-gram profile can have a cardinality considerably larger than that of normal *pq*-grams. However, our similarity join methods aim at *data-centric* XML datasets which usually have strings of moderate length. Further, we can use DB accesses to obtain shorter strings in selected parts of a *document-centric* XML for similarity evaluation on content and structure; in the remainder, the evaluation is done on structure only (see Sect. 5.2).

Path-Grams. Additionally, we have explored another approach to combining structure and content information. For each text node, we generated the normal set of q-grams and appended the root-to-leaf path of the containing element node. *Path-gram* (*PG*) denotes this method of composing information used for similarity computation. In an XDBMS environment, its evaluation can be supported by the documents path synopsis [9] which delivers such path information for free. Finally, because the path-gram generation is performed on the basis of each text node in isolation, we (conceptually) extended each string by prefixing and suffixing it with q-1 *null characters* (a null character is a special symbol not present in any string) in a way similar to [6].

5 Tree Similarity Joins and Their XDBMS Integration

A tree similarity join needs to locate the root nodes of the candidate subtrees to be compared. For this reason, we use XPath or XQuery expressions which declaratively specify two sets of nodes defining both sides of the join operands. For example, a tree similarity join (*TSJ*) can be used to validate incoming tree structures against an *assume-to-be-correct* (*atbc*) reference data source. Intuitively, we could denote such an operation by $(X)TSJ(Y)$ where X specifies the reference structure (e.g., *doc*(*atbc.xml*)/*atbc*/*article*[*reviewer* = *J.Gray*]) and Y the subtrees to be correlated (e.g., *doc*(*pub.xml*)/*pub*/*paper*).

We have designed a family of tree similarity joins, where we focused on the combination of structural and textual similarity. Hence, TSJ_{EPQ} and its versions $v1 - v3$ jointly consider tokenized text nodes and element nodes using *epq*-grams (TSJ_{v1-3} for short). Furthermore, TSJ_{PG} exploits *path*-gram tokens in a similar way. In contrast, TSJ_{CS} independently evaluates textual and structural similarity, which can be achieved in either sequence. The final result is the (weighted) average of each evaluation.

5.1 TSJ Processing

The various phases of XDBMS-based join processing are illustrated in Fig. 4. *Subtree access* indicates that the subtrees qualified for *TSJ* have to be identified and fetched from their disk-based storage locations to a memory-resident working area for further processing. This aspect is more than essential, because, in large XML documents, inappropriate selection of subtrees to be checked may consume the largest fraction of the overall response time for a similarity join. Moreover, repeated subtree access may become necessary if the intermediate token sets are too large to be kept in memory. Hence, it is of particular importance that document scans can be avoided and that index-based scans minimize physical disk accesses. In addition, support of predicates based on node type and subtree paths is also required (see Sect. 5.2).

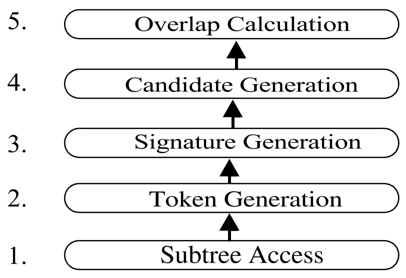
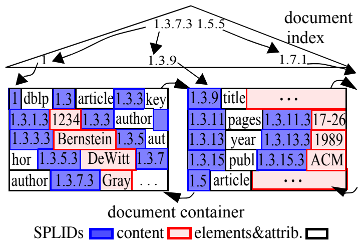


Fig. 4. Phases of TSJ processing

Token generation is essentially determining the quality of the overall *TSJ* processing (see Sect. 6.2). For each qualified subtree, token sets in the form of q-grams, pq-grams, or epq-grams have to be generated whose sizes depend on



document index
1.3.7.3 1.5.5
1
1.3.9
1.7.1
1 dblp 1.3 article 1.3.3 key
1.3.1.3 1234 1.3.3 author
1.3.3.3 Bernstein 1.3.5 aut
hor 1.3.5.3 DeWitt 1.3.7
author 1.3.7.3 Gray . . .
1.3.9 title . . .
1.3.11 pages 1.3.11.3 17-26
1.3.13 year 1.3.13.3 1989
1.3.15 publ 1.3.15.3 ACM
1.5 article . . .
document container
SPLIDs content elements&attrib.

Fig. 5. Stored XML document

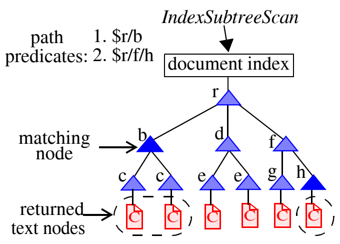


Fig. 6. Path search arguments

the underlying structure and content of the tree and the tokenization method itself (their fundamentals are sketched in Sect. 4). The primary goal of *signature generation* is to filter the potentially large token sets and to derive a compact representation facilitating candidate generation. A signature scheme such as prefix filter considerably reduces the original set of tokens; for example using the (unweighted) Jaccard similarity with a threshold of 0.9, the obtained signature has a cardinality of roughly 10% of the original token set. In the forth phase, *candidate generation* applies to the sets of signatures delivered by both join operands. Using equi-joins, this method selects those signature sets sharing at least one element. Hence, candidate pairs are prepared for the final processing phase, where *overlap calculation* delivers quantities to which similarity metrics can be directly applied. Depending on threshold values, some pairs of subtrees are satisfying the similarity condition. The nodes of the matching subtrees are then forwarded to the output structure.

5.2 Integration into XTC

As a testbed for XML research, we have developed XTC as a prototype XDBMS exhibiting full DB capabilities [10]. Because similarity join functionality is domain-related and optimal results strongly depend on application characteristics, we did not deeply integrate the *TSJ* operator. Instead we implemented this operator using a plug-in at a higher layer of abstraction while, however, exploiting the efficient XDBMS core functionality such as indexes, scans, etc., to access and extract disk-based XML fragments and provide them for *TSJ* processing.

For all storage structures, we heavily rely on B*-trees as the proven base structure guaranteeing balanced trees, logarithmic accesses, and dynamic reorganization. Most important for fine-granular and flexible processing is the node labeling scheme used for XML trees. A substantial amount of empirical experiments with various schemes led us to the conclusion that prefix-based labeling is superior to all competitor schemes [8]. In XTC, we implemented (after less convincing experiments with other schemes) a variant of prefix-based node labeling, called SPLID scheme (Stable Path Labeling IDentifier).

In Fig. 5, we have illustrated the storage structure of a sample XML document, where the B*-tree is composed by the *document index* and the *document container* as a set of doubly-chained pages. Because the document is stored in document order, the SPLIDs in the container pages lend themselves to prefix compression which typically reduces the storage space needed for SPLIDs to 20% of the original size [8]. Furthermore, element and attribute names are replaced by means of a vocabulary of VocIDs (consisting of two bytes). We can optionally apply content compression based on Huffman codes, thus resulting in a space-economical native XML storage representation. In addition to the document index needed to directly access structure or text nodes via their SPLIDs, we provide a variety of indexes for element/attribute or path access, content search, or even CAS queries (content&structure). All of them are based on B*-trees and use lists of SPLIDs for document access. For example, an element index would deliver for value *author* the locations for all author elements (1.3.3, 1.3.5, 1.3.7, ...), whereas a content index would provide the locations of the indexed text values in the document, e.g., 1.3.7.3, ... for Gray.

The most important access mechanism for *TSJ* is the *IndexSubtreeScan* [12]. With a reference list of SPLIDs, typically delivered from a suitable index, it locates the related nodes for the join operand via the document index. Then, the respective subtrees are scanned and their nodes are delivered to the token generation process. Optionally, IndexSubtreeScan may take a set of predicates to return selected parts of a subtree. Predicates based on node type are used to retrieve only structural nodes or textual nodes from a subtree. A more complex type of predicate is the so-called *path predicate* where a path expression is used to locate specific nodes of subtrees and then only the text nodes attached to these nodes are delivered for token set generation. Note that all structural nodes are returned, regardless of whether they are contained in the matching nodes or not. Fig. 6 illustrates an example with two path predicates. Such predicates are used by instances of TSJ_{v1-3} to avoid long strings in document-centric XML.

6 Experiments

After having introduced the algorithms and their system embedding, we are ready to present our experimental results. The main goal of our evaluation is to comparatively measure performance and accuracy of the *TSJ* family.

```
<!ELEMENT imdb (movie)>
<!ELEMENT movie (title, production_year?,
                 cast*, crew*)>
<!ELEMENT title          (#PCDATA)>
<!ELEMENT production_year (#PCDATA)>
<!ELEMENT cast    (actor*, actress*)>
<!ELEMENT name     (#PCDATA)>
<!ELEMENT role    (#PCDATA)>
<!ELEMENT note     (#PCDATA)>
<!ELEMENT actor    (name,role?,note?)>
<!ELEMENT actress  (name,role?,note?)>
<!ELEMENT crew     (producer*, director*)>
<!ELEMENT producer (name, type?)>
<!ATTLIST producer type CDATA >
<!ELEMENT director        (name, note?)>
```

Fig. 7. DTD of the IMDB dataset

We start with three well-known datasets: *DBLP* containing computer science publications, *IMDB* (www.imdb.com) storing information about movies, and *NASA* presenting astronomical data. *DBLP* and *NASA* are already available in XML format. For the *IMDB* dataset, we generated an XML document, whose DTD is shown in Fig. 7. Statistics describing all datasets are given in Table 1. (We consider only article subtrees in *DBLP*.) *DBLP*

Table 1. Dataset statistics

dataset	#sub-trees	avg nodes/ subtrees	distinct tags/ paths	max/avg path length	structure /content ratio	avg node string size	max string size	avg tree string size
DBLP	328838	24	26/37	6/3.02	1.083	19.99	665	239.65
IMDB	380000	56	13/14	5/4.88	1.64	12.9	224	277.85
NASA	2435	371	69/78	8/7.76	1.43	33.22	14918	5069.31

and *IMDB* show data-centric characteristics whereas *NASA* is more document-centric. We then produce "dirty" copies of these datasets by performing controlled transformations on content and structure. We implemented a program for error injection in XML datasets, which enabled the variation of "dirtiness" by specifying parameters: percentage of duplicates to which transformations are applied (*erroneous duplicates*) and extent of transformations applied to each erroneous duplicate (*error extent*). Injected errors on text nodes consist of word swappings and character-level modifications (insertions, deletions, and substitutions). Structural modifications consist of node insertion and deletion, position swapping, and relabeling of nodes. Insertion and deletion operations follow the semantics of the tree edit distance algorithm [15]. Swapping operations replace the whole subtree under the selected node, while relabeling only changes the nodes name (with a DTD-valid substitute). Frequency of modifications (textual and structural) is uniformly distributed among all nodes of each subtree.

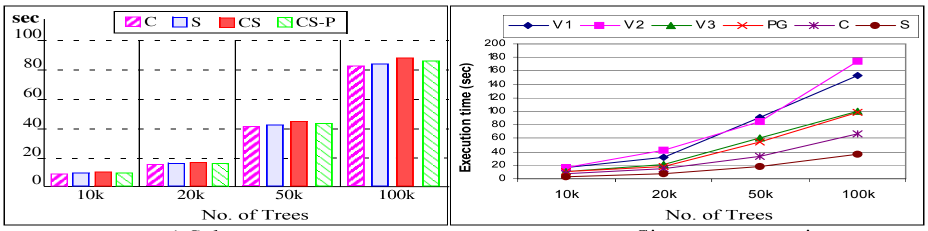
In all evaluations, we used the IDF-weighted Jaccard similarity. The ordering of the prefix-filter signature elements is defined by also using IDF weights $w(t)$ specified as $w(t) = 1 + log\left(\frac{|T_1|+|T_2|}{ft}\right)$, where ft is the total number of subtrees in T_1 and T_1, which contain t as a token. We use q-grams of size 2 for text nodes and pq-grams with p of size 2 and q of size 2 for structural nodes. We observed best results with this setting, especially in the accuracy experiments. All tests were run on an Intel Pentium IV computer (two 3.2 GHz CPUs, 1GB main memory, 80GB external memory, Java Sun JDK 1.6.0) as the XDBMS server machine where we configured XTC with a DB buffer of 250 8KB-sized pages.

6.1 Performance Results

In the first experiment, we analyze and compare execution time and scalability of our similarity join algorithms and their suboperator components, subtree scan and signature generation, for an increasing number of input trees which were selected from *IMDB* and *DBLP* datasets[1]. We only report the results for *IMDB*, since the results for *DBLP* are consistently similar to them in all experiments.

Subtree scan operators are the main component of the XTC system embedding. Therefore, we can properly observe the integration effects of TSJ into the

[1] *NASA* has a richer structure, however, TSJ_{PG} and TSJ_{CS} do not support path predicates, which is necessary for queries against document-centric XML.



a) Subtree scan b) Signature generation

Fig. 8. Suboperator performance results

XTC architecture by analyzing it separately. Fig. 8*a* compares the execution time of the 4 types of tree scan access used by our operators: content only (C), structure only (S), and content and structure without and with path search arguments (*CS* and *CS-P*, respectively). The search argument used is the path expression $/movie/cast/author$. All tree scan operators perfectly scale with the input dataset size, which emphasizes that we have achieved a smooth embedding of these operators into the XTC architecture. Furthermore, there is no significant performance variation among the suboperators. Finally, the path search argument used by *CS-P* does not present any impact on performance.

Fig. 8*b* shows the results of the prefix-filter signature generation time of the three versions of *epq*-grams, *path*-gram, and the two components of TSJ_{CS}, content-only (q-grams) and structure-only (*pq*-grams). Again, all suboperators scale very well with the input size. The relative results reflect almost exactly the token set sizes produced by each tokenization method. Since the token sets have to be ordered during the prefix-filter computation, larger sets cause higher overhead. This fact is emphasized by the best performance of the signature generation for *pq*-grams, which produces the shortest token sets, even though the algorithm used for textual token set generation is much simpler. Furthermore, the calculation of *pq*-gram sets is entirely performed using SPLIDs which are highly optimized in XTC and, consequently, does not negatively impact the performance. The signature generation of *path*-gram is slower than that of q-gram, because it operates on extended strings resulting in larger token sets.

The results for the complete TSJ evaluation of are depicted in the Fig. 9. For this experiment, we used datasets containing 50% of erroneous duplicates. The error extent was configured to be 30%, i.e., the percentage of erroneous structural and textual nodes in duplicate subtrees. We use a threshold fixed at 0.85. TSJ_{CS} is evaluated by first evaluating text nodes and afterwards structural nodes; this processing sequence was superior in our experiments. In addition, TSJ_{CS}

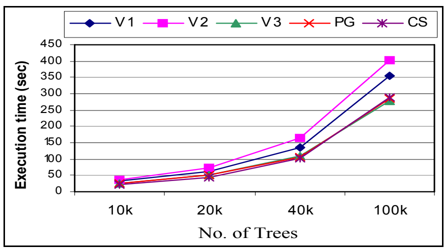


Fig. 9. Full *TSJ* evaluation

is evaluated in conjunctive mode, where only the subtrees returned by the textual operator are considered by the structural operator.

We observe that all operators scale and do not present dramatic performance variations. TSJ_{PG}, TSJ_{v3}, and TSJ_{CS} are superior and very close, whereas TSJ_{v1} and TSJ_{v2} are 15%–25% slower. Because they produce larger token sets, the number of signature collisions is higher, thereby requiring more similarity evaluations. Finally, we mention that the size of the input dataset determines only partially the execution time. In addition, the number of similar tree pairs occurring in the experiment significantly influences the execution time required.

6.2 Accuracy Evaluation

We now evaluate the quality of our similarity operators. For this experiment, we generate each dataset by first randomly selecting 500 subtrees from the original dataset and then generating 4500 duplicates from them (9 per subtree). As query workload, we randomly select 100 subtrees from the above dataset. For each queried input subtree T, the trees T_R in the result returned by our similarity join are ranked according to their calculated similarity with T. In this experiment, we do not use any threshold parameter, and therefore the rank reported is *complete*. Further, during data generation, we keep track of all duplicates generated from each subtree; those duplicates form a partition and carry the same identifier.

We use well-known evaluation metrics from Information Retrieval to evaluate the quality of our methods: the *non-interpolated Average Precision* (AP), the *maximum F1 score*, and the interpolated precision at recall levels 0.0, 0,1,...,1,0. AP is $\frac{1}{\#relevanttrees} \times \sum_{r=1}^{N} [P(r) \times rel(r)]$, where r is the rank, N the total 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. The F1 measure is the harmonic mean of precision and recall over the ranking. The interpolated precision at recall r is the highest precision found for recall levels higher than r. We report the mean of the AP and F1 measure over the query workload.

We applied the TSJ_{CS} operator using the weights 0.5 (0.5) and 0.25 (0.75) for structural (textual) similarity score in the weighted average calculation and represented them in our experimental charts as *CS-0.5* and *CS-0.25*, respectively. Following a strategy similar to [1], we classify our test datasets into *dirty*, *moderate*, and *low* error datasets according to the parameters used in the data generation as shown in Table 2. Errors in duplicate subtrees (i.e., the error extent) are applied to structural nodes at the same rate as to the textual nodes. For example, L1 contains 10% of alteration on its text nodes as well as 10% of

Table 2. Duplicate dataset classes

Class	Name	Percentage of	
		erroneous duplicates	error extent
Low	L1	10	10
Low	L2	30	10
Moderate	M1	30	30
Moderate	M2	60	10
Dirty	D1	60	30
Dirty	D2	90	30
-	E1-E4	50	10-50

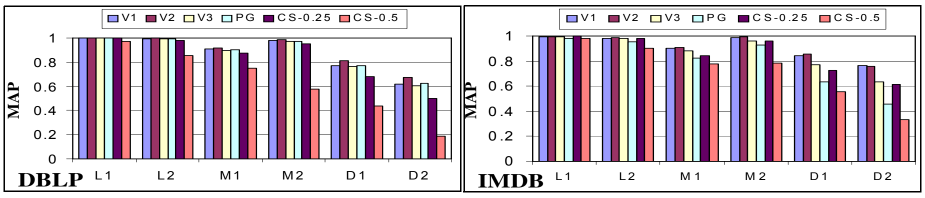


Fig. 10. MAP values for DBLP and IMDB datasets

alteration on its structural nodes. We also generated datasets containing only one specific type of structural error and with a fixed textual error extent to evaluate the effect of specific structural deviations.

Fig. 10 shows the mean AP (MAP) values for the classes of datasets generated from *DBLP* and *IMDB*. All operators perform well on the L1 dataset. The performance of *CS-0.5*, however, suffers a considerable degradation already on L2 and presents a very poor accuracy for dirty datasets. On the other hand, *CS-0.25* shows much more resilience to errors. In fact, its performance is comparable to the *epq*-gram-based operators for low and moderate data set classes. These results demonstrate the higher importance of textual tokens for the similarity evaluation. Because *text* tends to be more selective than structure, our results confirm the common notion that less frequent elements contribute more to similarity assessment. In general, the *epq*-grams operators outperformed all competitors, with TSJ_{v2} being the best and TSJ_{v3} being the worst among them. Clearly, the reduction of the number of appearances of each character node in version 3 (decreased by 1 achieved by pruning of *qnull*, see Sect. 4.2) negatively impacts the quality of the results.

The next experiment we report explores the sensitivity of our operators to specific types of structural deviations. We consider four types of errors: *Add nodes*, that inserts nodes along a root-to-leaf path, thereby increasing the depth of a subtree; *Tree-up* moves nodes up in a subtree modifying ancestor-descendant relationships; *Swap* changes the ordering of nodes (it may also move nodes to the sibling of its parent); and *Rename* that changes the nodes name. We do not include deletion of trees in this experiment, because this kind of error also (substantially) changes the content of a subtree and, therefore, would blur the results with text-related modifications. For these datasets, we restricted the erroneous duplicates to 50%, the textual error extent to 10%, and increased the structural error extent from 10% to 50% in steps of 10%.

The results are shown in Fig. 11. Our first observation pertains to the poor performance of TSJ_{PG} for all types of errors. Because this method relies on root-to-leaf information to generate its token set, it suffers large accuracy degradation as the structural errors increase. *CS-0.25* shows the best results for *Add Nodes* and *Rename* datasets. Indeed, they are practically not affected by these types of structural errors. Because the evaluation of textual and structural similarity is done independently and the structural score has less weight, in addition, the method is resilient to structural errors. However, a natural question that may arises is whether or not it is a good behavior for a similarity measure to report high scores

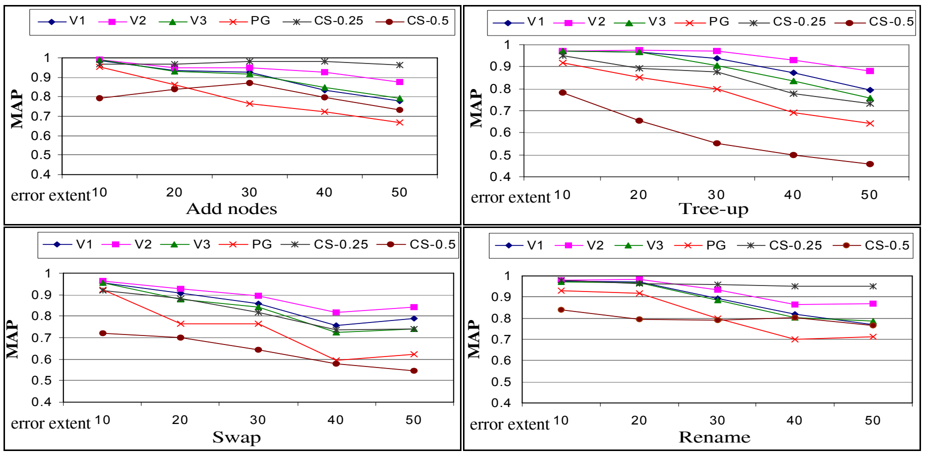


Fig. 11. MAP values for datasets with specific errors

for subtrees containing nodes having different tags. Note that this is similar to the case in our example in Fig. 1. Among the *epq*-gram-based operators, TSJ_{V2} again presents the best results. Because its underlying tokenization method neither produces q-grams of size 1 (in contrast to TSJ_{v1}) nor decreases the number of text nodes appearing in *epq*-grams (in contrast to TSJ_{v3}), TSJ_{v2} can better explore textual similarity to deliver more accurate results.

In our final experiment, we focussed on the *NASA* dataset which has a more document-centric flavour, with much larger string sizes in the content part than the previous datasets. For our experiment, we generated the same classes of datasets described in Table 2. However, we only evaluated the *epq*-grams-based operators, since TSJ_{CS} and TSJ_{PG} do not support path predicates. We used $//author$ and $/dataset/title$ as path predicates to obtain the textual content of the subtrees. The results shown in Fig. 12 are similar to those of the other datasets: *epq*-v2 has the best accuracy among the various versions of *epq*-grams. We note that, in general, the results of all operators are better than those for *DBLP* and *IMDB*. Besides the textual information obtained by the path predicates, the rich structure of the subtrees provides a large amount of information to be exploited by the similarity evaluation.

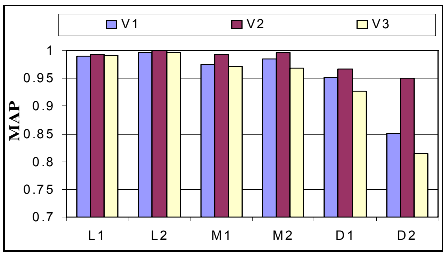


Fig. 12. NASA accuracy results

7 Conclusion

In this paper, we primarily explored an approach to tree-based similarity joins and analyzed its performance and accuracy when embedded in native XDBMSs.

Our results have shown that we achieved a seamless integration of similarity operators into XTC. The internal XDBMS processing, i.e., the specific support of our lower-level suboperators, enabled efficient evaluation of XML documents stored on disk thereby providing scalability in all scenarios considered. Our framework provides multiple instances of similarity joins which support different ways of tree similarity evaluation in a unified manner. We also explored a new concept, the so-called *epq*-grams, combining structural and textual information to improve the quality of similarity joins. Our results have shown that this technique considerably enhances the matching quality, i.e., the accuracy of the similarity join, especially on dirty datasets.

Acknowledgement: Work supported by CAPES/Brazil under grant BEX1129/04-0.

References

1. Amit, C., Hassanzadeth, O., Koudas, N., Sadoghi, M.: Benchmarking Declarative Approximate Selection Predicates. In: Proc. SIGMOD Conf., pp. 353–364 (2007)
2. Arasu, A., Ganti, V., Kaushik, R.: Efficient Set-Similarity Joins. In: Proc. VLDB Conf., pp. 918–929 (2006)
3. Augsten, N., Böhlen, M., Gamper, J.: Approximate Matching of Hierarchical Data using *pq*-Grams. In: Proc. VLDB Conf., pp. 301–312 (2005)
4. Chaudhuri, S., Ganjam, K., Kaushik, R.: A Primitive Operator for Similarity Joins in Data Cleaning. In Proc. ICDE Conf., p. 5 (2006)
5. Gravano, L., Ipeirotis, P., Jagadish, H., Koudas, S., Srivastava, D.: Text Joins in an RDBMS for Web Data Integration. In: Proc. WWW Conf., pp. 90–101 (2003)
6. Gravano, L., Ipeirotis, P., Jagadish, H., Koudas, M.S., Srivastava, D.: Approximate String Joins in a Database (Almost) for Free. In: Proc. VLDB Conf., pp. 491–500 (2001)
7. Guha, S., Jagadish, H., Koudas, N., Srivastava, D., Yu, T.: Integrating XML Data Sources using Approximate Joins. TODS 31(1), 161–207 (2006)
8. Härder, T., Haustein, M., Mathis, C., Wagner, M.: Node labeling schemes for dynamic XML documents reconsidered. DKE 60(1), 126–149 (2007)
9. Härder, T., Mathis, C., Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. In: Proc. IDEAS 2007, pp. 102–113 (2007)
10. Haustein, M.P., Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. DKE 61(3), 500–523 (2007)
11. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
12. Ribeiro, L., Härder, T.: Embedding Similarity Joins into Native XML Databases. In: Proc. 22nd Brasilian Symposium on Databases, pp. 285–299 (2007)
13. Sarawagi, S., Kirpal, A.: Efficient Set Joins on Similarity Predicates. In: Proc. SIGMOD Conf., pp. 743–754 (2004)
14. Ukkonen, E.: Approximate String Matching with *q*-grams and Maximal Matches. Theor. Comput. Science 92(1), 191–211 (1992)
15. Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and related Problems. SIAM Journal on Computing 18(6), 1245–1262 (1989)