SBBD 2007
XXII Simpósio Brasileiro de Banco de Dados

Embedding Similarity Joins into Native XML Databases

Leonardo Ribeiro, Theo Härder

University of Kaiserslautern, 67653 Kaiserslautern, Germany

{ribeiro|haerder}@informatik.uni-kl.de

Abstract. *Similarity joins in databases can be used for several important tasks such as data cleaning and instance-based data integration. In this paper, we explore ways how to support such tasks in a native XML database environment. The main goals of our work are: a) to prove the feasibility of performing tree similarity joins in a general-purpose XML database management system; b) to support string- and tree-based similarity techniques in a unified framework; c) to avoid relying on special data preparation or data structures to support similarity evaluation, such as partitioning or tailor-made index structures; d) to achieve a seamless integration of similarity operators and the existing database architecture.*

1 Introduction

Originally adopted as common syntax for data exchange, XML is increasingly used to directly represent pieces of information which are managed and maintained in a collaborative way. As a consequence, XML documents often have to be persistently stored in databases. As a result, large XML datasets are emerging for which transactional guarantees in multi-user environments such as integrity (data/relationship consistency) and concurrency control are required. Therefore, they depend upon similar maintenance tasks as their relational counterparts.

1.1 Important Application Areas

A basic maintenance procedure is *data cleaning* which deals with identifying and correcting data inconsistencies. A frequent form of inconsistency in data collections is the presence of multiple representations of a real-world entity. Among other inconveniences, the presence of such redundant information can erroneously inflate estimates of categories and, later, cause incorrect results in decision support queries. Even worse, values diverging from one another in redundant representations of the same entity may confuse consistency maintenance itself. Duplicate data arise due to instance level problems, such as misspellings or different naming conventions, and cannot be prevented by concepts like integrity constraints. The problem has been explored since the late 1950s by several research communities, including statistics, databases, natural language processing, and bioinformatics. In the database area, this problem is commonly referred to as the *fuzzy duplicate problem* to differentiate it from the standard duplicate problem, which considers two data instances to be duplicates if they are exact copies of one another. Along this paper, we use duplicate to mean *fuzzy duplicate* except when explicitly stated otherwise.

XML poses additional challenges to the duplicate elimination. Features such optional elements and attributes can result in document fragments with equal content but highly different structure. Moreover, the flexible nature of XML (and of the applications that use it) usually requires multiple or evolving schemas, too. Structural information may therefore be necessary to identify duplicates. We believe that such characteristics make XML datasets even more prone to the appearance of duplicates in comparison with relational ones.

1.2 Preliminaries and Related Work

An approach to tackle the duplicate problem is the use of *similarity joins*. In the relational model, a similarity join pairs tuples from two relations whose specified attributes are similar. The similarity of these attributes is expressed using a *similarity function* and a pair of tuples is qualified if the similarity function returns a value greater than a given threshold. In the XML model, the similarity join is applied to document fragments and the arguments of the similarity function are subtrees rooted at given elements that need to be matched [10]. Commonly used in data integration scenarios [6], a similarity join operator is also relevant for duplicate elimination, e.g., by using self-joins [2] or validating incoming data against *reference relations* [3]. An attractive aspect of similarity joins is the opportunity of exploiting query processing capabilities of the underlying database management system. Similarity joins are also referred to as *approximate joins*.

Guha et al. [10] propose a general but, to some extent, static framework for XML similarity joins using the notion of *tree edit distance.* Their optimization techniques primarily address the saving of distance computations, where upper and lower bounds for the tree edit distance are used as filters. Because each XML document is represented by a distance vector using a pivot-based approach, this method is less flexible, in particular, for dynamic documents. Our query-based framework, in contrast, easily copes with document updates, because it does not rely on pre-calculated distances. Furthermore, we refine our framework by embedding efficient XDBMS access operators and by evaluating the similarity join in a tailor-made XTC system environment.

The support of different similarity functions is an important requirement for a similarity join framework. Recent works have explored *set overlap* to generalize a variety of similarity functions in the context of similarity joins [6, 17]. Informally, specified fields or elements to be used for comparison are first converted to sets, e.g., mapping strings to sets by means of a token generation method, and then predicates are applied to measure the partial set overlapping according to a specified similarity function. This approach can be used to exploit several similarity functions, e.g., Jaccard coefficient and Hamming distance, as well as to apply effective filters for metrics such as string edit distance [9]. Chaudhuri et al. [5] implemented this concept, called therein *set similarity joins,* by composing standard relational operators in a pipelined fashion.

1.3 Our Contributions

To the best of our knowledge, approximation algorithms, in general, are hardly explored in the context of XML DBMSs, so far. As explained in the following, a similarity join operator is usually applied in a domain- or application-dependent way. Therefore, the individual algorithms used in such operators are highly application-specific and tailored towards the specific kinds of data such that their *deep integration* into the DBMS is not ad-

equate. Hence, we have to identify ways how to suitably integrate a collection of such approximation-supporting algorithms, how to select them for a specific application from a kind of framework, and how to combine or parameterize them for effective solutions. Our specific contributions in this work are:

- to support string- and tree-based approximation techniques for similarity joins in a unified framework
- to achieve a seamless integration of similarity operators and the existing database architecture while taking advantage of the given database internals to access the XML documents stored on disk and optimize the processing steps towards the final similarity result
- to prove the feasibility to efficiently process tree similarity joins in a general-purpose XML database; this aspect has to regard the behavior of all DBMS components in multi-user mode (e.g., locking as well as logging & recovery)
- to give some (initial) quantitative results by conducting empirical experiments and performance measurements using our prototype XML database system called XTC (XML Transactional Coordinator) [13].

This paper is organized as follows. In Section 2, we briefly sketch important definitions for our operators and similarity measures. Section 3 describes how similarity joins are performed at an abstract level independently from a concrete DBMS implementation whereas Section 4 sketches the sequence of steps to derive a join result. At this point, Section 5 introduces some internals of our XTC system, before we are able to outline the specific implementation of the similarity join operator in Section 6. We present our experimental work and the results gained in Section 7, before we wrap up the paper in Section 8.

2 Definitions

According to common practice, we model an XML document as an ordered labeled tree. Similarly to the DOM data model [8], we distinguish between element nodes and text nodes. We make no distinction however between element nodes and attribute nodes and consider each attribute as child of its owning element. We also disregard other node types such as Comment, CDATA, Entity, etc., and consider only data of string type. Finally, we assume a node labeling scheme that uniquely identifies all nodes in an XML collection and captures the containment relationship among the nodes in an XML document. Note that such a labeling scheme is an indispensable capability in a native XML database. See [12] for a comprehensive study.

2.1 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 the following result*:

$$\{(T_1, T_2) \in F_1 \times F_2 \,|\, (sim(T_1, T_2) \geq \gamma) \equiv TRUE\}$$

Note that we have defined the similarity function to be 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, for example, in case of a node set only consisting of text nodes. Recall that, if structure matters, the labeling scheme allows to identify containment relationships among a set of nodes.

2.2 Set-overlap-based Similarity Measures

Several similarity measures can be reduced to the problem of set overlap or multi-set[1] overlap [4, 17]. Given two sets representing two objects, different ways to measure their overlap raise various notions of similarity (or dissimilarity). There are several examples of such measures, among others Jaccard similarity, generalized edit distance [3], and Hamming distance. We observed that the method used to map an object to a set also has influence on the notions of similarity, since it can determine the properties of the object under consideration by the similarity measure. For example, given an XML tree, we can produce sets representing its textual or 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, hereafter, *set-overlap* measurement. In the following, we present the methods used to convert XML tree information to sets and to represent the set-overlap measurement in join predicates.

2.2.1 Mapping Text to Sets

When the objects of interest are represented by strings, e.g., a set of text nodes, the well-known approach is to convert them to a set of tokens. Tokens are normally considered as words or as q-grams, i.e., continuous substrings of size *q*.

***Example 1*:** Consider the strings “IBM Corporation” and “IBM Corproation”. They can be converted to the following 3-grams sets:

$s1$: {’IBM’, ’BM ’, ’M C’, ’ Co’, ’Cor’, ’orp’, ’rpo’, ’por’, ’ora’, ’rat’, ’ati’, ’tio’, ’ion’}
$s2$: {’IBM’, ’BM ’, ’M C’, ’ Co’, ’Cor’, ’orp’, ’rpr’, ’pro’, ’roa’, ’oat’,’ati’, ’tio’, ’ion’}

The method used for token generation influences another similarity measure properties such as the ability to capture misspellings (e.g. word-based vs. q-gram), result quality, and space overhead (e.g. different values for q in q-gram methods [9]).

2.2.2 Mapping Structure to Sets

We use the notion of *p-q grams* presented by Augstein et al. [1] to map the tree structural information to a set. Next, we briefly review the basic definitions of this method.

Given a tree T and integral values for $p > 0$ and $q > 0$, an extended tree $T^{p,q}$ is constructed from 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; q children to each leaf node. Figures 1(a) and (b) show tree T and its extended form $T^{2,3}$, respectively.

[1] In the following, we make no distinction between sets and multi-sets, referring to both concepts as sets.

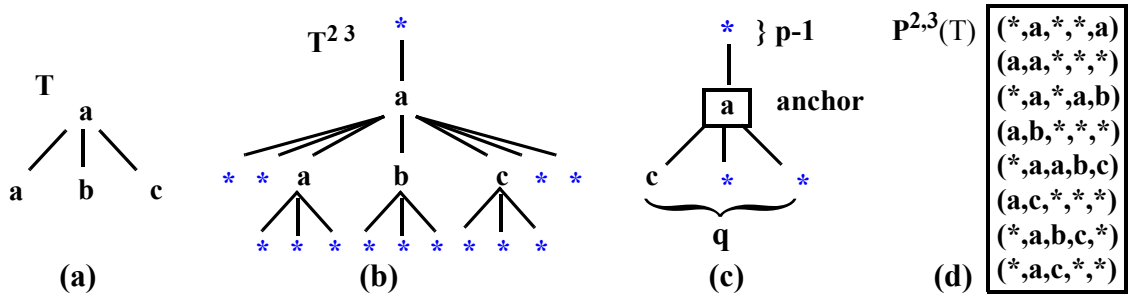

Figure 1: Steps of the generation of pq-gram tokens

Next, fixing an *anchor node a*, a pq-gram *g* of T is a subtree of $T^{p,q}$ composed of labels from the following nodes: p nodes $a_{p-1}, \ldots a_1, a$, that represents the p-part of *g*, where a_i is the ancestor of *a* at distance *i*; q contiguous children $c_i, \ldots, c_{i+q}$ of *a*, that represent the q-part of *g*. Figure 1(c) shows a *pq*-gram of $T^{2,3}$ in Figure1(b). Finally, a *pq-Gram Profile* is obtained by the labels of all *pq*-grams of an extended tree. Figure 1(d) shows the resulting *pq*-Gram Profile.

The set of *pq*-grams can be easily produced by a *preorder traversal* of the corresponding tree. In our case, the input is a stream of structural nodes received in *document order* by the underlying access mechanism (covered in Section 5). The resulting *pq*-Gram Profile can then be further manipulated as a set of tokens.

2.2.3 Set-overlap in Join Predicates

In [4], Chaudhuri et al. specify a general class of predicates to represent set-overlap-based similarity measures as join predicates:

$$pred(r, s) = \wedge_i(|r \cap s| \geq e_i)$$

In the predicate above, e_i is an expression involving constants and the cardinality of *r* and *s*. Note that the above predicate can be seen as a *zooming in* on the similarity join predicate of Definition 1.

In the following, we define a set-overlap-based similarity measure and give its representation as a similarity join predicate. For ease of exposition, we choose the Jaccard similarity that has a straightforward general predicate representation. We also will use it in other examples along this paper. We refer the reader to [4] for other examples of similarity functions used as set-overlap predicates.

Definition 2 (Jaccard Similarity): Let r and s be two sets. The Jaccard similarity, denoted by $jacc(r, s)$, is defined as:**.**

$$jacc(r, s) = \frac{|r \cap s|}{|r \cup s|}$$

Example 2: Consider the sets *s1* and *s2* in Example 1. We have $jacc(s1, s2) = 9/(13 + 13 - 9). \cong 0.53$.

Finally, a predicate of the form $jacc(r, s) \geq \gamma$ rewritten into the general class of predicates as follows:

$$|r \cap s| \geq \frac{\gamma}{1+\gamma}(|r| + |s|)$$

2.3 Signature Scheme

To avoid the bulky similarity evaluation for each pair of sets, a *signature scheme* is commonly used [2]. Given a collection of sets as input, a signature scheme produces a "summary" for each set, that roughly maintains their pairwise similarity according to a given measure. More specifically, for any two sets r and s, and their respective signatures $Sig(r)$ and $Sig(s)$, we have $Sig(r) \cap Sig(s) \neq \varnothing$ whenever $sim(r, s) \geq \gamma$. Likewise, two sets with similarity lower than a threshold do not share any signature. These properties can be approximately satisfied, i.e., with probabilistically defined bounds of *false positives* and *false negatives* [5]. A variety of parameters can be combined to construct the signature, such as thresholds, characteristics of input data sets, set cardinality, etc. Essentially, a signature scheme behaves like a filtering method. Next, we briefly review two methods that are currently being used in our framework.

2.3.1 Prefix-Filter

The prefix-filter signature scheme [4] 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 $c - \gamma + 1$ of r and s should intersect. Variations of this basic idea are used to handle weighted sets and normalized similarity functions. The ordering of the sets is picked to keep the elements with smallest frequencies in the prefix-filter signature, i.e., the elements are ordered by increasing order of their frequency in the data collection. This relation is commonly obtained by using the well-known *inverse document frequency* (or IDF) weighting scheme.

2.3.2 Size-based Filter

A size-based filter uses the intuitive notion that, if two sets have a similarity higher than some threshold, then their sizes should be within a same range. These ranges are defined in specific ways for different similarity measures. For example, for Jaccard similarity, given two sets r and s, the size-based filter is defined on the basis of the following ratio:

$$min\left(\frac{r}{s}, \frac{s}{r}\right) \geq \gamma$$

By rewriting this relationship, we have the following interval [17, 2]:

$$\gamma \leq \frac{|r|}{|s|} \leq \frac{1}{\gamma}$$

3 Tree Similarity Joins

We now delve from the abstract concept of Definition 1 into a concrete framework that allows us to find similarities among XML document fragments over a native XML database. A natural concern when dealing with duplicate identification on XML datasets is *structural heterogeneity*. While it is always possible to attempt to use solely the textual information to find duplicates, the structure of XML documents may carry valuable information to support the final classification of a pair as duplicate or non-duplicate. Another obvious reason to consider structural similarity is to avoid erroneous matching of documents with syntactically similar textual information, but represented by unrelated concepts, i.e., elements with different tags but same content. Therefore, it is highly desirable in a framework for tree similarity joins to evaluate both textual and structural similarity.

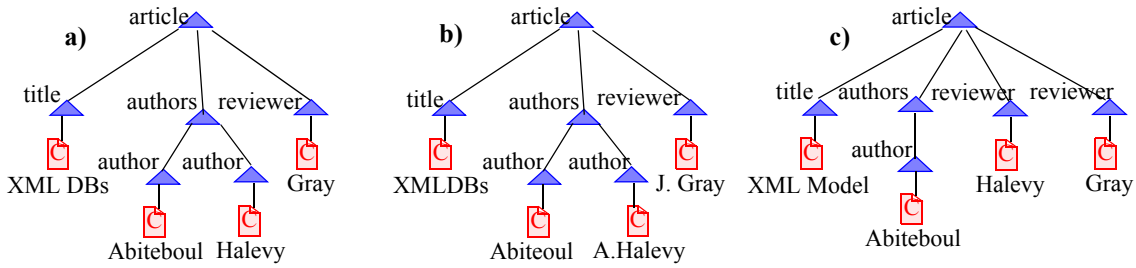

Figure 2: Example XML document fragments

Consider the simple examples in Figure 2. An application trying to match the document fragments a) and b) could use the textual similarity to classify them as duplicate candidates. In addition, the fact that a) and b) have the same structure can provide further evidence for the final classification. Consider now a situation where a) being compared to c). Again the textual similarity provides an indicator of similarity. However, c) apparently refers to another article. Hence in this example, the evaluation of the structural similarity could help to classify the two document fragments as non-duplicates.

Our tree similarity join (or *TSJ)* can be used to pair trees based on their kind of similarity—structural, textual, or both. Therefore, it is convenient to further distinguish TSJ_S, which evaluates structural similarity, and TSJ_T, which evaluates textual similarity. TSJ_S uses only element nodes for similarity calculation, whereas TSJ_T uses only text nodes. We consider a second variant of TSJ_S, denoted by TSJ_{SF}, in which text nodes are used jointly with element nodes in the structural similarity evaluation. This flavor of structural evaluation can be useful if textual data quality is not a concern. In Section 7, we will see that TSJ_{SF} has considerable performance benefits. In the following, we use TSJ_S to address both methods of structural similarity evaluation.

The evaluation of both kinds of similarity, structural and textual, is conceivable and can be realized in differing ways. Here, we derive its results by a sequential combination of TSJ_T and TSJ_S, which we denote by TSJ_{TS}.

All similarity joins classified above use set-overlap predicates as defined in Section 2.2.3. We indicate the related predicate by superscripts. For example, TSJ_s^{jacc} denotes the structural evaluation using Jaccard as a similarity predicate. TSJ_{TS} uses two such predicates connected by a Boolean operator. Note that the evaluation of TSJ_{TS} resembles methods based on *multi-attribute similarity*. In the relational scenarios, multi-attribute methods use more than one attribute of a relation participating in the match operation. The results of these multiple evaluations are then combined by a *merging function* [10] which gives the overall similarity score. Here, we do not consider the use of merging functions in TSJ_{TS}, but such a feature is orthogonal to our framework and could be easily added.

The starting point for a tree similarity join is the location of the root nodes of the qualified subtrees to be matched. Here, we use an XPath or XQuery expression which declaratively specify two sets of nodes to define both sides of the join operands. To enable flexibility as much as possible, we allow for separate subtree specifications, that is, the join partners can be specified via different path expressions.

Example 3: Consider a reference XML data source, say *dblp.xml*, containing only known-to-be-clean elements. Given a second file, *pub.xml*, we want to use *dblp.xml* to find duplicate *paper* elements related to a specific author, say 'Jim Gray'. A similarity join between

article elements in *dblp.xml* and *paper* elements in *pub.xml*, which evaluates both the structural and textual similarity using the Jaccard metrics, is represented by the following expression:

$$doc("dblp.xml")/dblp/article/[author='Jim\ Gray']TSJ_{ST}^{jacc}\ \ doc("pub.xml")/pub/paper$$

4 Course of Abstract *TSJ* evaluation

In this section, we describe the course of abstract *TSJ* evaluation in our framework. Implementation details are presented later in Section 6. The main steps of the overall procedure are shown in Figure 3. Basically, given a set of nodes as input, the following actions are executed: 1) the database is accessed and specified portions of the subtrees rooted at each input node are fetched; 2) a set of tokens is generated from 1; 3) a signature for each token set is produced; 4) candidate pairs are generated from the collection of tree signatures; 5) the final overlap calculation is performed and pairs of root nodes whose subtrees satisfy the similarity condition are output.

Subtree access is executed on top of the storage engine by the node manager. Here, we only discuss the main services required by our framework. In a subsequent section, when some details of our database architecture are introduced, we show how these requirements are met. Essentially, subtree access operations need to collect all the nodes structurally contained in the node specified as input. Support of predicates based on node type is also required, i.e., to retrieve only the structural or textual nodes from a subtree.

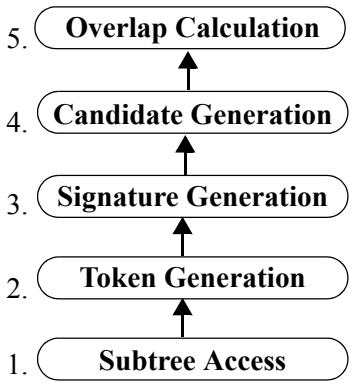

Figure 3: Course of TSJ evaluation

The following step, the token set generation, maps sets of nodes to a token set representation. The specific method is determined by the instance of *TSJ* being processed. For example, TSJ_T produces sets of n-grams and TSJ_S produces sets of *pq*-grams (see Section 2). Together with the subtree access, the method of token set generation fully determines the instance of *TSJ*.

The next steps are the same for all instances of *TSJ* and follow the lines of the general class of *set-similarity join*s [4]. Signatures of each set of tokens are produced using the techniques described in Section 2.3. Next, candidate pairs are generated by joining all sets that share at least a signature. Finally, an exact set-overlap calculation is performed on each candidate pair and those satisfying the similarity join condition are output.

5 Providing Processing Support by XTC

Specific similarity join functionality is domain-related and optimal results depend on application characteristics. Therefore, deep integration of such functionality into the DBMS kernel is not recommended. In contrast, only standard core functionality (indexes, scans, etc.) is used to extract data potentially contributing to similarity join results; join processing itself is performed using a kind of plug-in at a higher layer of abstraction.

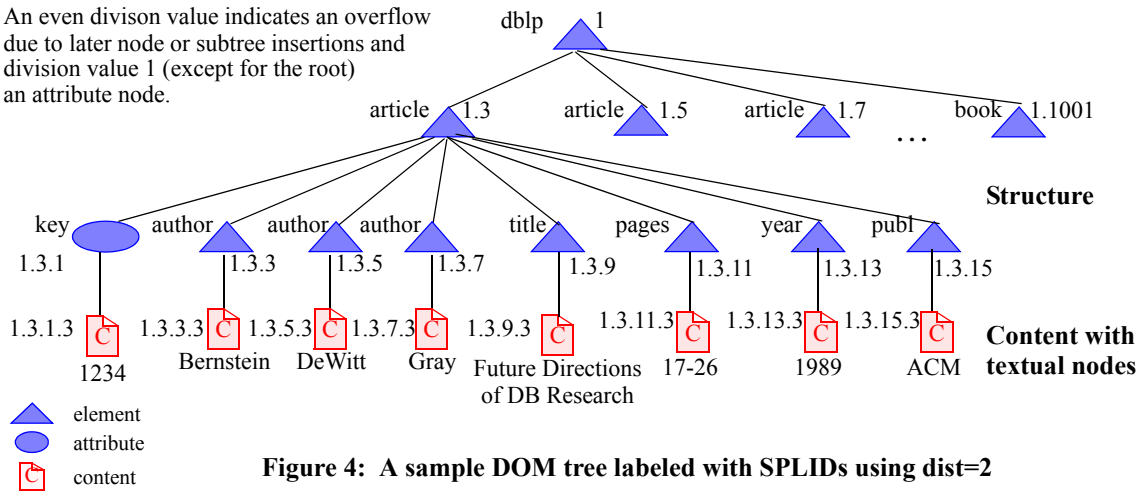

Figure 4: A sample DOM tree labeled with SPLIDs using dist=2

Similar to the relational model, joins are I/O intensive operations in the XML world. Typically, large numbers of XML subtrees have to be located on disk, fetched to memory, pre-processed and filtered by the tokenizer resp. *pq*-gram use, before extensive similarity-join comparisons can be executed on memory-resident data structures. DBMS integration further means to adequately support transactional multi-user operations which directly requires selective access to XML trees (as opposed to sequential scans) and fine-grained locking of minimal subtrees or node sets (instead of entire trees).

To provide efficient and effective solutions, space-economic storage structures and selective index-based access to XML documents are needed in the first place. For this reason, we have implemented a tailor-made native XML storage [13] which achieves drastically reduced space occupancy. Therefore, it guarantees minimized I/O cost. Let us sketch the key concepts of XML storage in XTC. In Figure 4, we have illustrated a fragment of the well-known and highly dynamic DBLP document using prefix-based node labels which implement the concept of Dewey order [12]. The abstract properties of Dewey order encoding—each label consists of so-called divisions (separated by dots in the external format) and allows to identify the path from the document's root to the resp. node and its local order w.r.t. the parent node; in addition, optional sparse numbering facilitates node insertions and deletions—are described in [12]. Refining this idea, a number of similar labeling schemes were proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism.

Our scheme refines Dewey order mapping: with a *dist* parameter used to increment division values, it may leave gaps in the numbering space between consecutive labels and introduces an overflow mechanism when gaps for new insertions are in short supply—a kind of adjustment to expected update frequencies. Since any prefix-based scheme such as OrdPath [16] or DeweyIDs [12] can avoid relabeling under subtree insertions and is functionally equivalent and appropriate for our document storage, we use the term SPLID (Stable Path Labeling IDentifier) as synonym for all of them. Each node essentially carries the labels of all ancestors up to the root of the tree. SPLIDs effectively support the evaluation of all XPath axes, can be used for indexing element/attribute nodes and are very helpful for the lock manager when setting intention locks along ancestor paths. Note that the SPLIDs stored in document order (left-most depth-first) lend themselves to prefix compression and are typically reduced to about 25% of the uncompressed format [12].

Consider the lenghty element names in Figure 4. Because we typically have a limited amount of names (<100), we can use a directory-based encoding (one byte) for them by so-called VocIDs in the storage and index representations. Document storage is based on variable-length files as document containers whose page sizes varying from 4K to 64K bytes could be configured to the document properties. We allow the assignment of several page types to enable the allocation of pages for documents, indexes, etc. in the same container. Efficient declarative or navigational processing of XML documents requires a fine-granular DOM-tree storage representation which easily preserves the so-called round-trip property when storing and reconstructing the document (i.e., the identical document must be delivered back to the client). Furthermore, it should be flexible enough to adjust arbitrary insertions and deletions of subtrees thereby dynamically balancing the document structure. Fast indexed access to each document node, location of nodes by SPLIDs, as well as navigation to parent/child/sibling nodes from the context node are important demands. As illustrated by Figure 5, we provide an implementation based on B*-trees which maintains the nodes stored in document order and which cares about structural balancing. While indexed access and order maintenance are intrinsic properties of such trees, some additional optimizations are needed. Variations of the entry layout for the nodes allow for single-document and multi-document stores, key compression, use of vocabularies, and specialized handling of short documents. As shown in Figure 5 by sketching the sample XML document of Figure 4—, a B-tree, the so-called *document index*, with key/pointer pairs (SPLID+PagePtr) indexes the first node in each page of the *document container* consisting of a set of chained pages. Using sufficiently large pages, the document index is usually of height 1 or 2. Because of reference locality in the B-tree while processing XML documents, most of the referenced tree pages are expected to reside in DB buffers—thus reducing external accesses to a minimum.

For an XML document, all structure and content nodes are stored in document order using a container as a set of doubly chained pages. The stored node format is of variable length and is composed of entries of the form (SPLID, name) or (SPLID, value). The content is further compressed using some Hufman encoding. Being an orthogonal issue and not important for similarity join processing, we do not discuss content compression in detail. In summary, our format stores the nodes carrying prefix-compressed SPLIDs and some administration data (2 bytes).

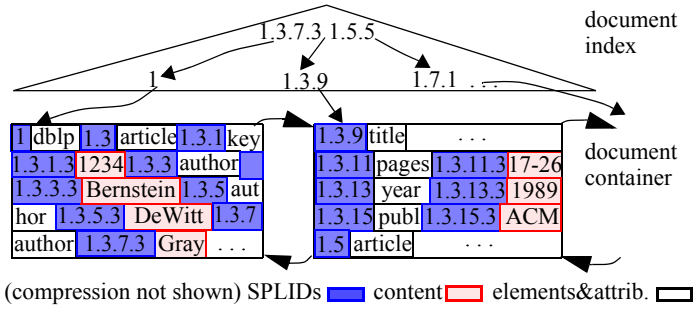

Figure 5: Stored XML document

In addition to the document store, various indexes may be created, which enable access via structure (element or attribute nodes) or content (values of leaf nodes). An element index consists of a name directory with (potentially) all element names

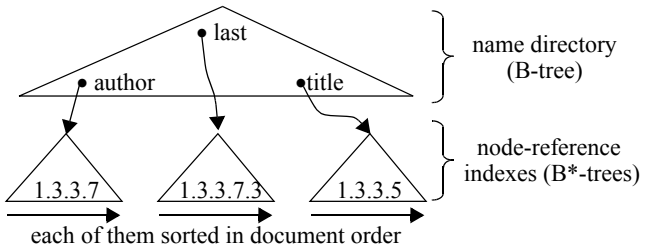

Figure 6: Organization of structure indexes

occurring in the XML document (Figure 6); this name directory often fits into a single page. Each specific element/attribute name refers to the corresponding nodes in the document store using SPLIDs. In case of short reference lists, they are materialized in the index; larger lists of references may, in turn, be maintained by a node reference index as indicated in Figure 6. Content indexes are created for root-to-leaf paths, e. g., /dblp/article/author, and again are implemented as B*-trees keeping for each indexed value a list of SPLIDs as references to the related locations in the document. When processing a query, a hit list of SPLIDs is built using one or several indexes. Then, the qualified nodes together with their related path instances are located via the document index (see Figure 5). In all cases, support of variable-length keys and reference lists is mandatory; additional functionality for prefix compression of SPLIDs is again very effective.

5.1 TreeScan vs. IndexSubtreeScan

A straightforward question that arises when fetching stored subtrees is about the best scan technique. There are two types of scans available in XTC. The first one called *TreeScan* traverses the whole XML document and only the subtrees of interest need to be identified and returned. The second type called *IndexSubtreeScan* uses the document index to find the qualified nodes for the join operand and then proceeds by scanning the subtree rooted at each node separately. Here, we discuss pros and cons of both approaches.

TreeScan only needs to traverse the document index and then perform sequential processing along the chained pages. Since we cannot rely on SPLIDs alone to identify subtrees boundaries, we have to look at node content to find the beginning of each subtree. This process touches all the pages of a document resulting in high overhead if the subtrees are sparse. A more subtle drawback is the handling of nested subtrees, i.e., if one subtree of interest contains another subtree of interest. It would require extra data structures to unnest subtrees adding considerable complexity to the scan processing. Finally, this operation requests a shared tree lock on the entire document.

IndexSubtreeScan traverses the document index for each element returned by the element index, which can be very expensive. On the other hand, there are favorable aspects which pay-off, at least partially, the expenses. First, since we jump to the beginning of each subtree using the document index, we only need to check the SPLIDs to identify the end of a subtree. Second, it is not necessary to take special care of nested subtrees. In this case, however, overlapping nodes will be retrieved more than once. Third, IndexSubtreeScan allows a smooth integration into our framework. As we will see in the next section, it can be easily plugged into our operators to produce tuples containing the subtree's root SPLID and its n-grams or *pq*-grams. Finally, in a multi-user environment, IndexSubtreeScan allows a finer lock granule, because only the subtrees of interest need to be locked. We observe, however, that a transaction could have to acquire so many locks that lock escalation is beneficial.

Obviously, the subtree selectivity determines the simple scan performance. In our experiments in XTC, we achieved better performance with IndexSubtreeScan even when 50% of the document were accessed. Anyway, the issues previously discussed for the TreeScan should not be ignored (handling of nested elements, control of subtree boundaries). A more accurate access path selection should also consider the underlying document structure, in addition to the selectivity. Currently, we only use IndexSubtreeScan in our framework. The definition of a cost model is a future task.

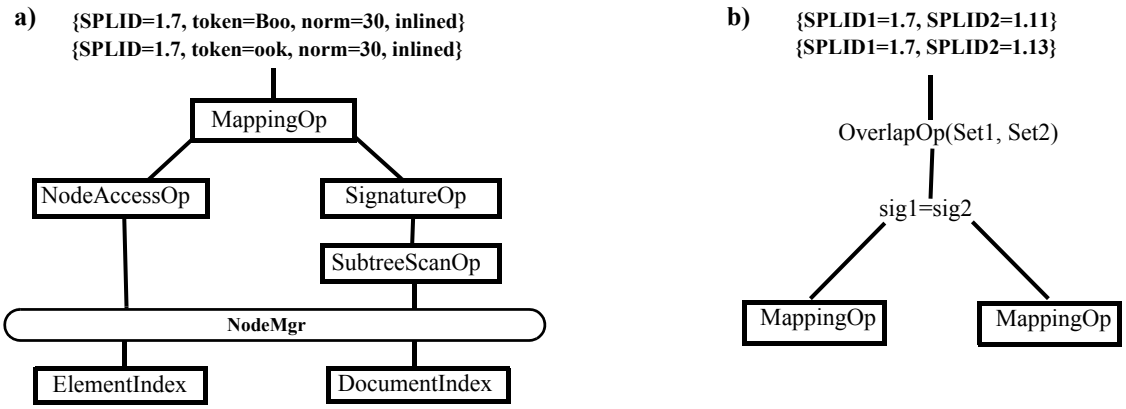

Figure 7: *TSJ* suboperators for similarity join processing

6 Implementation of the Similarity Join Operator

In this section, we discuss the details of the implementation of *TSJ* operator. The main design concern is the seamless integration of the *TSJ* framework into the XTC architecture. Performance is achieved by enabling pipelining as much as possible and avoiding redundant subtree scans. The execution plan of *TSJ* is composed of similarity-specific operators (e.g., SignatureOp), relational-based (e.g. equi-joins) and tuple-based operators (e.g., MappingOp). The latter type are physical implementations of an extended version of the XTC algebra [15]. Due to limited space, we only describe informally the semantics of such operators in the context of the whole execution plan.

The physical operator tree shown in Figure 7a is used to process one operand of a similarity join and implements the first three steps of the *TSJ* evaluation (see Section 4). Delivered by ElementIndex, NodeAccessOp extracts sequences of SPLIDs that qualify against some query predicate, and passes them on to MappingOp. This top-most operator in the query tree is in charge of producing sets of tuples containing individual SPLIDs (labeling the roots of qualified subtrees) together with signatures of the subtrees the SPLIDs point to. To compute these signatures, SignatureOp is called by MappingOp for each SPLID received, which, in turn, triggers a subtree scan providing the required subtree nodes via the DocumentIndex, that is, for each SPLID, a subtree scan (using an ONC cycle) has to access the XML document stored on disk. The nodes returned to SignatureOp are used to generate tokens, either q-grams or *pq*-grams, depending on the instance of *TSJ* being processed, before the signatures are derived. Finally, MappingOp outputs for each element in the signature set a tuple containing this element and the input SPLID (as illustrated in Figure 7a). In addition, depending on the similarity function used, the cardinality of the original token set is also added in the output tuple.

A known shortcoming of all signature-based algorithms is that the original token sets are lost when their signatures are created. However, the original token sets are required in the subsequent set-overlap calculation. Hence, we have two choices: to perform an additional subtree scan to re-generate the tokens or to attach them to each output tuple in an inlined representation [4]. In our implementation, we use the second option. We note that such a decision essentially represents a trade-off between space overhead and (substantial) performance gains. The inlined representation approach has shown its merits when the tokens sets are not unbounded (that is, they had to be stored on disk).

The remainder of the *TSJ* execution, depicted in Figure 7b, combines the results of both operand evaluations and represents a straightforward implementation of the needed concepts (Section 4). An equi-join is used to find all sets that share at least one signature. Because two trees can have more than one signature in common, *exact duplicate* pairs of matching candidates can be produced. Therefore, a final duplicate elimination is required, before Overlap outputs these pairs of SPLIDs.

7 Experiments

After having introduced the algorithms and their system embedding, we are ready to present our experimental results. All similarity measures supported by *TSJ* are well-known and so far some empirical and comparative studies are done in isolation, e.g., [1, 7]. The efficiency of the signature scheme used, prefix filter, has been analyzed as well, but not in an XML context [2, 4]. Therefore, we concentrate on XDBMS performance measurement of different instances of our *TSJ* implementation using varying data sizes.

7.1 Datasets and Setup

We start with the well-known and publicly available DBLP dataset containing computer science publications. Our DBLP version has about 260000 article nodes. The mean size of 3-gram sets generated from the articles nodes is about 235; the mean size of 23-gram sets (*pq*-grams) is about 25.

We first generate several initial datasets by randomly selecting articles from DBLP and combining them to separate XML *reference documents which* have varying sizes (5k, 10k, 20k, and 50k articles) to explore scalability issues. Using copies of these reference documents, we then generate modified datasets in a controlled way (called *duplicate datasets*) to enable textual and structural similarity evaluation, used to check performance of the similarity join. Hence, the similarity joins are performed between equal-sized XML files from the reference and duplicate datasets.

The duplicate datasets for textual evaluation was 'prepared' with artificially inserted textual errors in 50% of the article subtrees. These errors consisted of character insertions, deletions, and substitutions and were randomly distributed among all textual nodes of each subtree. The number of textual errors in each subtree ranges from 1 to 5 with a uniform probability of existence. In the second duplicate datasets, we accomplished structural modifications. The operations applied to the nodes consisted of empty node insertions, deletions, position swapping and relabeling of nodes. The frequency distribution of changes and error seeds was the same as used for the generation of textual duplicates.

In all evaluations, we used Jaccard similarity with threshold fixed at 0.85. The related token IDF weights needed for the prefix filter were calculated in a pre-processing step; the related overhead is not included in our results. The IDF weights are used to define the ordering of the prefix-filter signature elements, which are specified as follows: $\log\left(\frac{|T|+|T'|}{f_t}\right)$, where f_t is the total number of subtrees T and T', which contain t as token.

All tests were run on an Intel XEON computer (four 1.5 GHz CPUs, 2 GB main memory, 300 GB external memory, Java Sun JDK 1.6.0) as the XDBMS server machine. We configured XTC with a DB buffer of 250 8KB-sized pages.

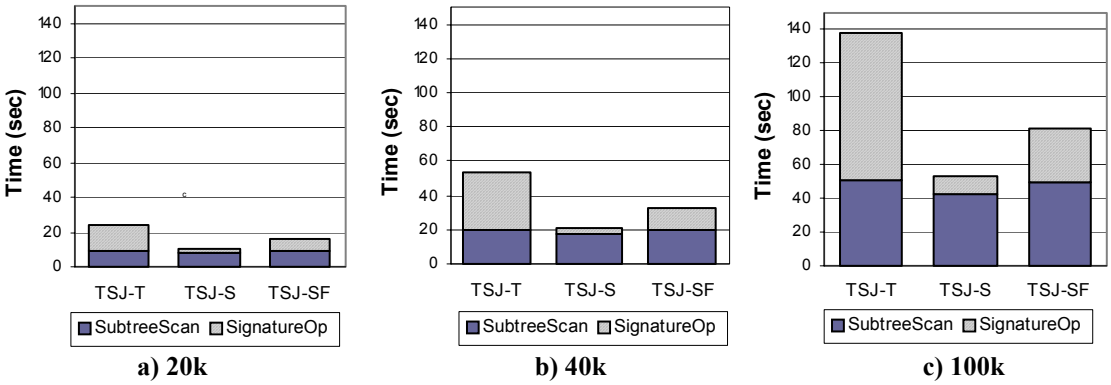

Figure 8: SubtreeScan and SignatureOp computation time

7.2 Results

We first analyse the lower-level suboperators of *TSJ in* isolation (see Figure 7a). Proceeding in this manner, we can properly observe the effects of the integration of *TSJ* into the XTC architecture. Figure 8 shows the computation time of SubtreeScan and SignatureOp for varying input dataset size. We use q-grams of size 3 to evaluate TSJ_T and pq-grams with p of size 2 and q of size 3 to evaluate TSJ_T and TSJ_{SF}. Our first observation is that the suboperators for all instances of *TSJ* scale with the input data set size in a perfect way. Interestingly, the SigGenOp is faster in TSJ_S than in TSJ_T even though the algorithm used for textual token set generation is simpler. The primary explanation of this effect is that the generated q-gram sets are bigger than the pq-gram sets. Since the token sets have to be ordered during the prefix-filter computation, it results in higher overhead. Moreover, the calculation of pq-gram sets is completely performed on the basis of SPLID node labeling which is highly optimized.

The results for the complete evaluation of *TSJ* are shown in Figure 9 where we took the same sizes of q-grams and pq-grams used in the previous experiment. We notice that TSJ_S scales poorly with the size of the input dataset (note, log scale is used on both axes). However, the size of the input dataset determines only partially the computation time. In addition, the number of similar tree pairs significantly influences the computation time required. Indeed, DBLP has a homogeneous structure and, even with modifications performed on the duplicate datasets, we still compare rather similar pairs of trees. On the other hand, the performance of TSJ_{SF} is not affected by the characteristics of the data in our experiments. TSJ_{SF} scales perfectly with the size of the input dataset and is roughly one order of magnitude faster than TSJ_T and two order of magnitude faster than TSJ_S. By using textual and structural nodes, TSJ_{SF} produces very selective pq-grams. Further, using IDF ordering, these pq-grams will be present in the prefix-filter signature. As a result, fewer pairs of trees pass through the filter and the size of the matching candidate set is reduced.

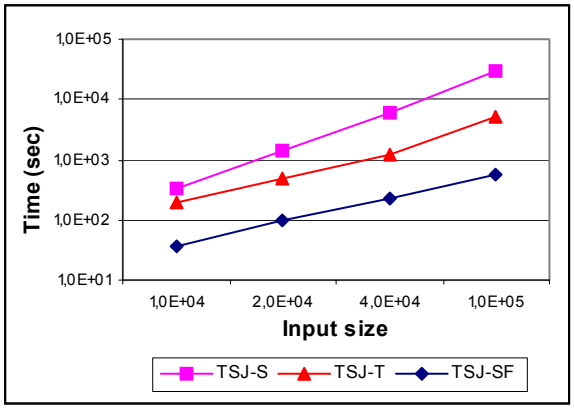

Figure 9: Complete STJ evaluation

8 Conclusion

In this paper, we presented a framework for similarity assessment of tree-structured documents in an XDBMS where we took advantage of the given database internals to optimize the various processing steps. Our framework provides multiple instances of similarity joins which allow varying ways of tree similarity evaluation in a unified way. Our results have shown that we achieved seamless integration of similarity operators into XTC where the internal DBMS processing, i.e., the specific support of our lower-level subooperators, could be used to efficiently evaluate XML documents stored on disk thereby providing scalability. The use of SPLIDs was particularly beneficial for the generation of token sets and the subtree scans to selectively access the qualified XML data to be compared. We revealed that various *TSJ* operators can have very different performance behavior on the same data sets. In particular, we have shown that TSJ_{SF} is two orders of magnitude faster than TSJ_{S}.

Our future work will include exploring the trade-offs provided by various *TSJ* operators between performance and quality of similarity results achieved.

Acknowledgment: This work was supported by CAPES/Brazil under grant BEX1129/04-0.

References

1. Augsten, N., Böhlen, M., and Gamper, J. (2005). Approximate Matching of Hierarchical Data using *pq*-Grams. In Proc. VLDB Conf., pp. 301-312.
2. Arasu, A., Ganti, V., and Kaushik, R. (2006). Efficient Set-Similarity Joins. In Proc. VLDB Conf., pp. 918-929.
3. Chaudhuri, S., Ganjam, K., Ganti, V., and Motwani, R. (2003). Robust and Efficient Fuzzy Match for Online Data Cleaning. In Proc. SIGMOD Conf., pp. 313-324.
4. Chaudhuri, S., Ganjam, K., and Kaushik, R. (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In Proc. ICDE Conf., pp. 5.
5. Cohen, E., Datar, M., Fujiwara, S., et al. (2000). Finding Interesting Associations without Supporting Prunning. In Proc. ICDE Conf., pp. 489-499.
6. Cohen, W. W. (1998). Integration of Heteroneous Databases without Common Domains using Queries Based on Textual Similarity. In Proc. SIGMOD Conf., pp. 201-212.
7. Cohen, W. W., Ravikumar, P. and Fienberg, S. (2003). A Comparison of String Distance Metrics for Name-Matching Tasks. In Proc. IJCAI-2003 Workshop on Information Integration on the Web.
8. W3C Document Object Model. http://www.w3.org/DOM/ (2007).
9. Gravano, L., Ipeirotis, P., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. (2001). Approximate String Joins in a Database (Almost) for Free. In Proc. VLDB Conf., pp. 491-500.
10. Guha, S., Jagadish, H. V., Koudas, N., Srivastava, D. and Yu, T. (2006). Integrating XML Data Sources using Approximate Joins. In Transactions on Database Systems. 31:1, pp. 161-207.
11. Guha, S., Koudas, N., Marathe, A., and Srivastava, D. (2004). Merging the Results of Approximate Match Operations. In Proc. VLDB Conf., pp. 636-647.
12. Härder, T., Haustein, M., Mathis, C., and Wagner, M. (2007). Node labeling schemes for dynamic XML documents reconsidered. In Data & Knowledge Engineering 60:1, pp. 126-149, Elsevier.
13. Haustein, M. P. and Härder, T. (2007). An Efficient Infrastructure for Native Transactional XML Processing. In Data & Knowledge Engineering 61:3, pp. 500-523, Elsevier.
14. Koudas, N., Marathe, A., and Srivastava, D. (2004). Flexible String Matching against Large Databases in Practice. In Proc. VLDB Conf., pp. 1078-1086.
15. Mathis, C. (2007). Integrating Structural Joins into a Tuple-Based XPath Algebra. In Proc. BTW Conf., pp. 242-261.
16. O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., and Westbury, N. (2004). ORDPATHs: Insert-Friendly XML Node Labels. In Proc. SIGMOD Conf., pp. 903-908.
17. Sarawagi, S. and Kirpal, A. (2004). Efficient Set Joins on Similarity Predicates. In Proc. SIGMOD Conf., pp. 743-754.