Second International Workshop on

Database Technologies for Handling XML information on the Web (DataX'06)

held at EDBT 2006
10th International Conference on
Extending Database Technology

Munich, Germany
March 26, 2006

Preface

This volume contains papers selected for presentation at the Second International Workshop on Database Technologies for Handling XML Information on the Web (DataX'06), held in conjunction with the International Conference on Extending Database Technology (EDBT'06) in Munich, (Germany) on March 26, 2006.

According to the main theme of the EDBT conference, today's horizons of the database field appear without limits. Peer-to-peer architectures, the Grid, personal information systems, pervasive and ubiquitous computing, networked sensors, biomedical informatics, virtual digital libraries, semantic virtual communities, database services, and trust management are just a few samples of the great challenges that drive research and development of the next generation of database technology. XML seems to be one of the main means towards this new generation. For these reasons we organized this workshop to give the opportunity to debate new issues and directions for the XML and database community in all these environments.

In response to the call for papers, 35 high quality submissions were received. Each paper was carefully reviewed by at least three members of the program committee and external reviewers. As result of this process, 10 papers have been selected for the workshop, covering a large variety of topics, ranging from document querying and updating to XML biology, security and schema mapping. In addition, Prof. H.V. Jagadish accepted our invitation to discuss new and interesting topics in XML and database research. A revised version of the workshop papers will be included in the joint EDBT'06 post-workshop proceedings published by Springer-Verlag, in the LNCS series.

We would like to thank the invited speaker, the external reviewers, and the program committee members for their efforts in the realization of this workshop. Finally, we wish to express our gratitude to the EDBT Organization and the EDBT Workshop Chair, Torsten Grust, for their support in all the workshop preparation as well as for printing the workshop proceedings.

DataX'06 Organizing Committee

Barbara Catania	DISI - University of Genova (Italy)
Akmal B. Chaudhri	IBM developerWorks (UK)
Giovanna Guerrini	DISI - University of Genova (Italy)
Marco Mesiti	DICO - University of Milano (Italy)

DataX'06 Program Committee

DataX'06 External Reviewers

Table of Contents

The Need for an Algebra in Practical XML Query Processing

H.V. Jagadish

University of Michigan

Abstract. Native XML database management has now come of age, and there are a number of systems that can be used to store and query XML, with ever improving scale and efficiency. However, most of these systems are implemented in a rather ad hoc manner.
In this talk, I will distill from our experiences with the Timber system, built at the University of Michigan, and argue that it is crucial to have an algebraic basis for constructing a complex data management system.

Hash-Based Structural Join Algorithms

Christian Mathis and Theo Härder

University of Kaiserslautern**

Abstract. Algorithms for processing Structural Joins embody essential building blocks for XML query evaluation. Their design is a difficult task, because they have to satisfy many requirements, e. g., guarantee linear worst-case runtime; generate sorted, duplicate-free output; adapt to fiercely varying input sizes and element distributions; enable pipelining; and (probably) more. Therefore, it is not possible to design *the* structural join algorithm. Rather, the provision of different specialized operators, from which the query optimizer can choose, is beneficial for query efficiency. We propose new hash-based structural joins that can process unordered input sequences possibly containing duplicates. We also show that these algorithms can substantially reduce the number of sort operations on intermediate results for (complex) tree structured queries (twigs).

1 Introduction

Because XML data is based on a tree-structured data model, it is natural to use path and tree patterns for the search of structurally related XML elements. Therefore, expressions specifying those patterns are a common and frequently used idiom in many XML query languages and their effective evaluation is of utmost importance for every XML query processor. A particular path pattern—the *twig*—has gained much attention in recent publications, because it represents a small but frequently used class of queries, for which effective evaluation algorithms have been found [1, 3, 7, 11, 14, 16].

Basically, a twig, as depicted in Fig. 1, is a small tree, whose nodes n represent simple predicates p_n on the content (text) or the structure (elements) of an XML document, whereas its edges define the relationship between the items to match. In the graphical notation, we use the double line for the descendant and the single line for the child relationship. For twig query matching, the query processor has to find all possible embeddings of the given twig in the queried document, such that each node corresponds to an XML item and the defined relationship among the matched items is fulfilled. The result of a twig is represented as an ordered[1] sequence of tuples,

a) **Queries**

```
Q1) //book[title="XML"]//author[.//city="Rome"]/name
Q2) for $b in //book, $a in $b//author
    where $b//title="XML" and $a//city="Rome"
    return ($a, $a/name)
```

b) **Twig for Q1 and Q2**

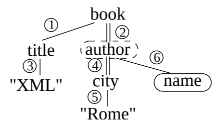


Fig. 1. Sample Query and Twig

** Database and Information Systems, D-67653 Kaiserslautern, Germany. {mathis|haerder}@informatik.uni-kl.de

[1] Here, "ordered" means: sorted in document order from the root to the leaf items.

where the fields of each tuple correspond to matched items. Usually, not all nodes of a twig generate output, but are mere (path) predicates. Therefore, we use the term *extraction point* [7] to denote twig nodes that do generate output (the boxed nodes in Fig. 1).

1.1 Related Work

For twig query matching, a large class of effective methods builds on two basic ideas: the *structural join* [1] and the *holistic twig join* [3]. The first approach decomposes the twig into a set of binary join operations, each applied to neighbor nodes of the twig (for an example, see Fig. 2). The result of a single join operation is a sequence of tuples S_{out} whose degree (number of fields) is equal to the sum of the degrees of its input tuples from sequences S_{inA} and S_{inB}. S_{out} may serve as an input sequence for further join operations. In the following, we denote the tuple fields that correspond to the twig nodes to join as the *join fields*. The underlying structural join algorithms are interchangeable and subject to current research (see the discussion below).

In [3], the authors argue that, intrinsic for the structural join approach, intermediate result sizes may get very large, even if the final result is small, because the intermediate result has to be unnested. In the worst case, the size of an intermediate result sequence is in the order of the product of the sizes of the input sequences. To remedy this drawback, twig join algorithms [3, 7] evaluate the twig as a whole, avoiding intermediate result unnesting by encoding the qualifying elements on a set of stacks.

Of course, holistic twig join algorithms are good candidates for physical operators supporting query evaluation in XDBMSs. However, they only provide for a small fraction of the functionality required by complete XPath and XQuery processors (e. g., no processing of axes other than *child* and *descendant*; no processing of order-based queries). Therefore, the development of new structural join algorithms is still valuable, because they can act as complemental operators in case the restricted functionality of twig joins is too small, or as alternatives if they promise faster query evaluation.

Existing structural join approaches can roughly be divided into four classes by the requirements they pose on their input sequences: A) *no requirements* [8, 11, 14]; B) *indexed input* [16], C) *sorted input* [1, 10, 16]; D) *indexed and sorted input* [4]. Especially for classes C and D, efficient algorithms have been found that generate results in linear time depending on the size of their input lists. In contrast, for class A, there is—to the best of our knowledge—no such algorithm. All proposed approaches either sort at least one input sequence [11], or create an in-memory data structure (a heap) requiring $O(nlog_2n)$ processing steps [14]. By utilizing hash tables that can be built and probed in (nearly) linear time, the algorithms we introduce in this paper can remedy this problem. Note, the strategies in [11, 14] elaborate on partition-based processing schemes, i. e., they assume a small amount of main memory and large input sequences, requiring their partition-wise processing. Their core join algorithm, however, is main-memory–based, as ours is. Therefore, our new join operators can be—at least theoretically[2]—combined with the partitioning schemes proposed in these earlier works. Answering

[2] [14] uses a perfect binary tree (PBiTree) to generate XML identifiers. In real-world scenarios, we assume document modifications that can hardly be handled with PBiTrees. Therefore, we

twig (and more complex queries) using binary structural join algorithms imposes three non-trivial problems: selecting the best (cheapest) join order (*P1*) to produce a sorted (*P2*) and duplicate-free (*P3*) output. P1 is tackled in [15], where a dynamic programming framework is presented that produces query executions plans (QEPs) based on cost estimations. The authors assume class C (and D) algorithms, which means that even intermediate results are required to be in document order on the two join fields. As a consequence, sort operators have to be embedded into a plan to fulfill this requirement. Consider for example the twig in Fig. 1. Let the circled numbers denote the join order selected by an algorithm from [15]. Then, three sort operators have to be embedded into the QEP (see[3] Fig. 2). Sort operators are expensive and should be avoided whenever possible. With structural join algorithms not relying on a special input order—like those presented in this paper—we can simply omit the sort operators in this plan. However, a final sort may still be necessary in some cases.

Problem P3 was studied in [8]. The authors show that duplicate removal is also important for intermediate results, because otherwise, the complexity of query evaluation depending on the number of joins for a query Q can lead to an exponential worst-case runtime behavior. Therefore, for query evaluation using binary structural joins, tuplewise duplicate-free intermediate result sequences have to be assured after each join execution. Note, due to result unnesting, even a (single) field in the tuple may contain duplicates. This circumstance is unavoidable and, thus, we have to cope with it. Because duplicate removal—like the sort operator—is an expensive operation, it should be minimized. For example in [6], the authors present an automaton that rewrites a QEP for Q, thereby removing unnecessary sort and duplicate removal operations. Their strategy is based on plans generated by normalization of XPath expressions, resulting in the XPath core language expressions. However, this approach does not take join reordering into account, as we do. Our solution to P3 is a class of algorithms that do not produce any duplicates if their input is duplicate free.

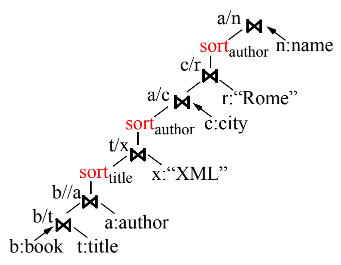


Fig. 2. Sample Plan

1.2 Contribution

We explore the use of hash-based joins for path processing steps of XML queries and identify the selectivity ranges when they are beneficial. In particular, we propose a class of hash-based binary structural join operators for the axes *parent*, *child*, *ancestor*, *descendant*, *preceding-sibling*, and *following-sibling* that process unordered input sequences and produce (unordered) duplicate-free output sequences. Furthermore, we show by extensive tests using the XTC (XML Transaction Coordinator)—our prototype of a native XDBMS—that our approach leads to a better runtime performance than sort-based schemes.

The remainder of this paper is organized as follows: Sect. 2 briefly describes some important internals of XTC, namely our node labeling scheme and an access method for

used SPLIDs (Sect. 2.1) instead. As a consequence, this “gap” had to be bridged to support the proposed partition schemes with our ideas.

[3] An arrow declares the input node of a join by which the output is ordered, where important. Possible are *root to leaf*, e. g., between “book” and “title”, and *leaf to root*, e. g., the final join.

element sequences. Sect. 3 introduces new hash-based algorithms. In Sect. 4 we present our quantitative results before we conclude in Sect. 5.

2 System Testbed

XTC adheres to the well-known layered hierarchical architecture: The concepts of the storage system and buffer management could be adopted from existing relational DBMSs. The access system, however, required new concepts for document storage, indexing, and modification including locking. The data system available only in a slim version is of minor importance for our considerations.

2.1 Path Labels

Our comparison and evaluation of node labeling schemes in [9] recommends node labeling schemes which are based on the Dewey Decimal Classification [5]. The abstract properties of Dewey order encoding—each label represents the path from the documents root to the node and the local order w. r. t. the parent node; in addition, sparse numbering facilitates node insertions and deletions—are described in [13]. Refining this idea, similar labeling schemes were proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of these schemes are ORDPATH [12], DeweyID [9], or DLN [2]. Because all of them are adequate and equivalent for our processing tasks, we prefer to use the substitutional name *stable path labeling identifiers* (SPLIDs) for them.

Here we only summarize the benefits of the SPLID concept which provides holistic system support. Existing SPLIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. Comparison of two SPLIDs allows ordering of the respective nodes in document order, as well as the decision of all XPath axis relations. As opposed to competing schemes, SPLIDs easily provide the IDs of all ancestors to enable direct parent/ancestor identification or access. This property is very helpful for navigation and for fine-grained lock management in the XML documents. Finally, the representation of SPLIDs, e. g., label 1.3.7 for a node at level 3 and also used as an index reference to this node, facilitates the application of hashing in our join algorithms.

2.2 Accessing Ordered Element Sequences

A B*-tree is used as a document store where the SPLIDs in inner B*-tree nodes serve as fingerposts to the leaf pages. The set of doubly chained leaf pages forms the so-called document container where the XML tree nodes are stored using the format (SPLID, data) in document order. Important for our discussion, the XDBMS

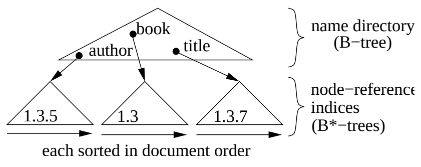


Fig. 3. Element Index

creates an *element index* for each XML document. This index consists of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 3). For each specific element name, in turn, a *node-reference index* is maintained which addresses the corresponding elements using their SPLIDs. Note, for the document store and the element index, prefix compression of SPLID keys is very effective because both are organized in document order directly reflected by the SPLIDs [9].

The leaf nodes in our QEPs are either element names or values. By accessing the corresponding node reference indexes, we obtain for them ordered lists of SPLIDs and, if required lists of nodes in document order by accessing the document store.

3 Hash-Based Structural Join Algorithms

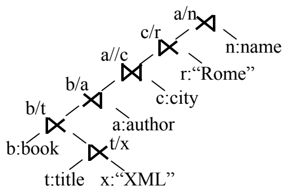


Fig. 4. Plan for Query 1

To be able to compete with existing structural join algorithms, we had to design our new algorithms with special care. In particular, the use of semi-joins has several important benefits. The processing algorithms become simpler and the intermediate result size is reduced (because the absolute byte size is smaller and we avoid unnesting). Several important design objectives can be pointed out:

Design single-pass algorithms. As in almost all other structural join proposals, we have to avoid multiple scans over input sequences.

Exploit extraction points. With knowledge about extraction points, the query optimizer can pick semi-join algorithms instead of full joins for the generation of a QEP. For example, consider the plan in Fig. 4 which embodies one way to evaluate the twig for the XPath expression in Fig. 1. After having joined the *title* elements with the content elements “XML”, the latter ones are not needed anymore for the evaluation of the rest of the query; a semi-join suffices.

Enable join reordering. Join reordering is crucial for the query optimizer which should be able to plan the query evaluation with any join order to exploit given data distributions. As a consequence, we need operators for the reverse axes *ancestor* and *parent*, too (e. g., the semi-join operator between *title* and “XML” in Fig. 4 actually calculates the parent axis).

Avoid duplicate removal and sort operations whenever possible. By using only algorithms that do not generate duplicates and operate on unordered input sequences, the query optimizer can ignore these problems. However, the optimizer has to ensure the correct output order, requiring a final sort operator. In some cases, this operator can be skipped: If we assume that the element scans at the leaf nodes of the operator tree in Fig. 4 return the queried element sequences in document order (as, for example, our element index assures), then, because the last semi-join operator is simply a filter for name elements (see Sect. 3.1), the correct output order is automatically established.

Design dual algorithms that can hash the smaller input sequence. The construction of an in-memory hash table is still an expensive operation. Therefore, our set of algorithms should enable the query optimizer to pick an operator that hashes the smaller of both input sequences and probes the other one, yielding the same result.

Table 1. Classification of Hash-Join Operators

	Output		
Hashed	*ancestor/parent*	*descendant/child*	*full join*
	Class 1: UpStep	Class 2: TopFilter	Class 3: FullTopJoin
parent	//**a**[b]	//**a**/b	//**a**/b, //**a**[b]
	ParHashA	ChildHashA	ChildFullHashA
ancestor	//**a**[.//b]	//**a**//b	//**a**//b, //**a**[.//b]
	AncHashA	DescHashA	DescFullHashA
	Class 4: BottomFilter	Class 5: DownStep	Class 6: FullBottomJoin
child	//a[**b**]	//a/**b**	//a/**b**, //a[**b**]
	ParHashB	ChildHashB	ChildFullHashB
descendant	//a[.//**b**]	//a//**b**	//a//**b**, //a[.//**b**]
	AncHashB	DescHashB	DescFullHashB

3.1 Classification of Algorithms

We can infer three orthogonal degrees of freedom for structural hash-join algorithms: the *axis* that has to be evaluated (*parent*/*child*/*ancestor*/*descendant*); the *mode* of the join (*semi*/*full*); and the choice of which input sequence to *hash* (*A* or *B*)[4]. The following naming scheme is used for our operators: `<axis> + <mode> + <hash>:` {`Par|Child|Anc|Desc`} {`Semi|Full`} `Hash`{`A|B`} ("Semi" is omitted for brevity). For example, the join operator between *title* and "XML" in Fig. 4 is a `ParHashB` operator, because it calculates the parent axis, is a semi-join operator, and hashes the sequence of possible children.

For an overview of all possible operators refer to Table 1: The column header defines the input to be hashed, whereas the row header defines the output. For clarification of the semantics, each operator is additionally described by an XPath expression where the input sequence to hash is marked in bold face. The names of the operator classes describe the evaluation strategy of the join. They will be discussed in the following. Note, class 1–3 algorithms are dual to class 4–6 algorithms, i. e., they calculate the same result as their corresponding algorithms, but hash a different input sequence.

3.2 Implementation

To abstract from operator scheduling and dataflow control, we let all operators act in the same operating system thread and use the well-known iterator-based *open-next-close* protocol as a basis for the evaluation. Each algorithm receives two input sequences of tuples, where, due to intermediate result unnesting, duplicates on the join fields have to be expected.

All proposed algorithms in this paper consist of two phases. In phase one, a hash table `ht` is constructed using the join field of the tuples of one input sequence (either sequence A or B). In phase 2, the join field of the other input sequence is probed against `ht`. Depending on how a result tuple is constructed, the operators can be assigned to one of the six classes: *Full*Join* operators return a sequence of joined result tuples just as earlier proposals for structural join algorithms (e. g., [1]). Note, the qualifiers "Top" and "Bottom" denote which input sequence is hashed. The remaining classes contain

[4] Note, in the following, *A* denotes the sequence of possible ancestors or parents (depending on the context), whereas *B* denotes descendants or children.

```
Input: TupSeq A,B, Axis aixs, bool hashA
Output: TupSeq results,Local:HashTable ht

1  // phase 1: build hash table
2  if (hashA)
3    foreach (Tuple a in A)
4      hash a.jField() in ht;
5  else if (axis is ‘Par’ or ‘Child’)
6    foreach (Tuple b in B)
7      hash b.jField().parent() in ht;
8  else if (axis is ‘Anc’ or ‘Desc’)
9    List levelOcc = getLevels(A);
10   foreach (Tuple b in B)
11     foreach (level in levelOcc)
12      hash b.jField().anc(level) in ht;
13
14 // phase 2: probe
15 foreach (Tuple t in ((hashA) ? B : A)
16   if (! hashA and
17      t.jField() in ht) results.add(t);
18   else if (axis == ‘Child’ or ‘Par’)
19      if (t.jField().parent() in ht)
20       results.add(t);
21
22   else if (axis == ‘Desc’ or ‘Anc’)
23     List levelOcc = getLevelsByProb(A);
24     foreach (level in levelOcc)
25       if (t.jField().anc(level) in ht)
26         results.add(t);
27         break inner loop;
28
29 function hashEnqueue
30       (SPLID s, Tuple t, HT ht)
31   Queue q = ht.get(s);
32   q.enqueue(t);
33   hash (s, q) in ht;
34
35 function hashDelete (SPLID s, HT ht)
36   Queue q = ht.get(s);
37   foreach (Tuple t in q)
38     results.add(t);
39   ht.delete(s);
40
41 function hashFull
42       (SPLID s, Tuple current, HT ht)
43   Queue q = ht.get(s);
44   foreach (Tuple t in q)
45     results.add(new Tuple(t, current));
```

Fig. 5. *Filter Operator and Auxiliary Functions for *Step and Full*Join

semi-join algorithms. **Filter* operators use the hash table, constructed for one input sequence to filter the other one, i. e., tuples are only returned from the probed sequence. **Step* operators work the other way around, i. e., they construct the result tuples from the hashed input sequence.

**Filter Operators* (see Fig. 5): In phase one, for `ChildHashA` and `DescHashA`, the algorithm simply hashes the SPLID of the elements of the join fields (accessed via method `jField()`) into `ht` (line 4). Then, in phase two, the algorithm checks for each tuple *t* in B, whether the parent SPLID (line 19 for `ChildHashA`) or any ancestor SPLID (line 25 for `DescHashA`) of the join field is contained in `ht`. If so, *t* is a match and is appended to the result. Actually, for the descendant operator, we had to check all possible ancestor SPLIDs which could be very costly. To narrow down the search, we use the meta-information, at which levels and by which probability an element of the join field of A occurs (line 23). This information can be derived dynamically, e. g., when the corresponding elements are accessed via an element index scan, or kept statically in the document catalog.

The strategy for `ParHashB` and `AncHashB` is similar, with the difference, that in the hash phase the algorithm uses the join fields of input B to precalculate SPLIDs that might occur in A (lines 7 and 12). Again for the descendant operator, we use the level information (line 9), but this time the probability distribution does not matter. In the probing phase it only has to be checked, whether the current join field value is in `ht`.

Obviously, the output order of the result tuples is equal to the order of the probed input sequence. Furthermore, if the probed input sequence is tuplewise duplicate free, the algorithm does not produce any duplicates. The *hashed* input sequence may contain duplicates. However, these are automatically skipped, whereas collisions are internally resolved by the hash table implementation.

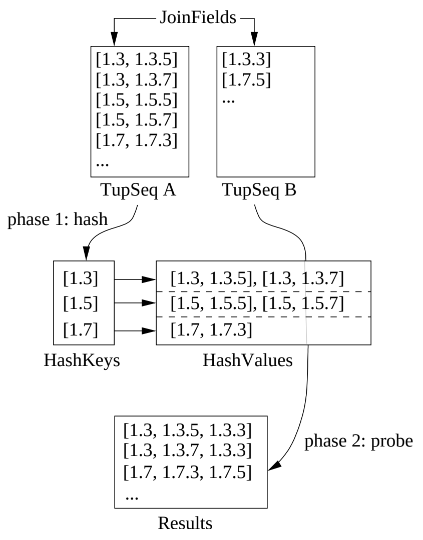
**Step Operators* conceptually work in the same way as their corresponding **Filter* operators. However, they do not return tuples from the probed, but from the hashed input sequence. Accordingly, tuples that have duplicates on the join field (e. g., TupSeq A of Fig. 6a) may not be skipped (as above) but have to be memorized for later output. The new algorithms work as follows: In the hash phase, the function `hashEnqueue()` (Fig. 5 line 29) is called instead of the simple hash statements in lines 4, 7, and 12). The first argument is the SPLID s of the join field (or its *parent*/*ancestor* SPLID). Function hashEnqueue() checks for s whether or not an entry is found in hash table `ht` (line 31). If so, the corresponding value, a queue q, is returned to which the current tuple is appended (line 32). Finally, q is written back into the hash table (line 33).

In the probing phase, we substitute the hash table lookup and result generation (lines 17, 19–20, 25–26) with the `hashDelete()` method (Fig. 5 line 35). For the given SPLID s to probe, this method looks up the corresponding tuple queue in the hash table and adds each contained tuple t to the result. Finally, the entry for s and its queue are removed from the hash table, because the result tuples have to be returned exactly once to avoid duplicates. The sort order of these algorithms is dictated by the sort order of the input sequence used for probing. If the hashed input sequence did not contain any duplicates, the result is also duplicate free.

*Full*Join Operators* resemble the **Step* operators. The only difference is the result generation. While **Step* algorithms are semi-join operators that do not produce a joined result tuple, *Full*Join* operators append the current result tuple with all tuples matched (as depicted in method `hashFull()`, Fig. 5 line 41). Note, opposed to `hashDelete()`, in `hashFull()` no matched entries from `ht` are deleted. For a brief *full join* example see Fig.6a: input sequence A for the `ChildFullHashA` operator is hashed on join field 1, thereby memorizing tuples with duplicates in the related queues. Then, the tuples from sequence B are probed against the hash table. For each match, each tuple in the queue is joined with the current tuple from B and appended to the result.

Space and Time Complexity. The space complexity (number of tuples stored) and time complexity (number of hashes computed) of the operators depend on the axis to be evaluated. Let $n = |A|$ and $m = |B|$ be the sizes of the input sequences. For the *parent*/*child* axis, the space and time complexity is $O(n+m)$. For the *ancestor*/*descendant* axis, the height h of the document also plays a role. Here the space complexity for classes 1–3 is also $O(n+m)$, whereas the time complexity is $O(n+h*m)$ (for each tuple in sequence B up to h hashes have to be computed). For classes 4–6, both space and time complexity are $O(n+h*m)$.

Beyond Twig Functionality: Calculation of Sibling Axes. With hash-based schemes and a labeling mechanism enabling the parent identification, the *preceding-sibling* and the *following-sibling* axes are—in contrast to holistic twig join algorithms—computable, too. Due to space restrictions, we can only show filtering algorithms, corresponding to the **Filter* classes above: In phase 1 operators `PreSiblHashA` and `FollSibl-HashA` (see Fig. 6b) create a hash table `ht` to store key-value pairs of *parent*/*child* SPLIDs. For each element in A, parent p is calculated. Then the following-sibling



```
Input: TupSeq A, B, Axis aixs
Output: TupSeq results, Local:HashTable ht

1  // phase 1: build hash table
2  foreach (Tuple a in A)
3    checkAndHash(a.jField(), axis)
4
5  // phase 2: probe
6  foreach (Tuple b in B)
7    SPLID s = ht.get(b.parent());
8    if( (axis == 'PreSibl' and
9         b.jField().isPreSibl(s)) or
10       (axis == 'FollSibl' and
11        b.jField().isFollSibl(s)) )
12     results.add(b);
13
14 function checkAndHash(SPLID a, Axis axis)
15   SPLID s = ht.get(a.parent());
16   if( (s is NULL) or
17       (axis == 'PreSibl' and
18        not s.isPreSibl(a)) or
19       (axis == 'FollSibl' and
20        not s.isFollSibl(a)) )
21     ht.put(a.parent(), a);
```

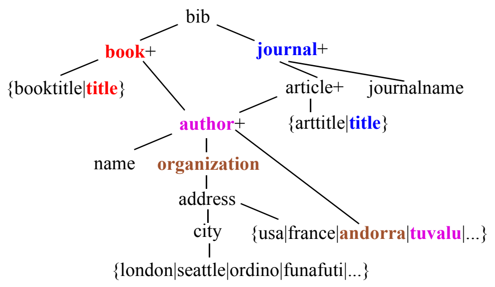
Fig. 6. a) Full*Join Example and b) Sibling Operator

(preceding-sibling) axis is evaluated as follows: For each parent SPLID *p*, the smallest (largest) child SPLID *c* in A is stored in `ht`. This hash table instance is calculated by successive calls to the `checkAndHash()` method (lines 14 to 21). While probing a tuple *b* of input B, the algorithm checks whether the SPLID on the join field of *b* is a following-sibling (preceding-sibling) of *c*, that has the same parent (lines 6 to 12). If so, the current *b* tuple is added to the result. Clearly, these algorithms reveal the same characteristics as their corresponding **Filter* algorithms: They do not produce any tuplewise duplicates and preserve the order of input sequence B.

4 Quantitative Results

To substantiate our findings, we compared the different algorithms by one-to-one operator comparison on a single-user system. 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 1.5.0) as the XDBMS server machine and a PC (1.4 GHz Pentium IV CPU, 512 MB main memory, JDK 1.5.0) as the client, connected via 100 MBit ethernet to the server.

To test the dependency between runtime performance and query selectivity, we generated a collection of synthetic XML documents, whose structure is sketched in Fig. 7. Each document has a size of 200 MB and contains bibliographic information. Because we were mainly interested in structural join operators for element sequences, the generated documents do not contain much text content. The schema graph is a directed acyclic graph (and not a tree), because an author element may be the child of either a book or an article element. We generated the documents in such a way, that we obtained the following selectivity values for the execution of structural joins between input nodes: 1%, 5%, 10%, 50%, and 100%. For example, for the query `//book[title]`,



*Filter Queries:
a) `//book[title]` **or** `//book/title`
b) `//journal[.//title]` **or** `//journal//title`

*Step and Full*Join Queries:
a) `//author[tuvalu]` **or** `//author/tuvalu`
b) `//organization[.//andorra]` **or** `//organization//andorra`

Fig. 7. Document Schema and Sample Queries

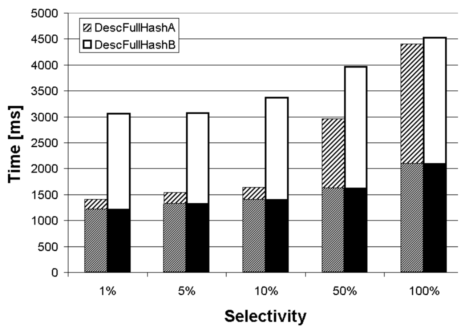
selectivity 1% means that 1% of all *title* elements have a *book* element as their parent (all others have the *article* element as parent). Additionally, we created 10% noise on each input node, e. g., 10% of all *book* elements have the child *booktitle* instead of *title*.

4.1 Join Selectivity Dependency of Hash-Based Operators

In a first experiment, we want to explore the influence of the join selectivities of the input sequences and, in case of varying input sequence sizes, their sensitivity on the hash operator performance. All operators presented in Table 1 revealed the same performance characteristics as a function of the join selectivity. Hence, it is sufficient to present an indicative example for which we have chosen the `DescFullHash*` operators. For the query `//journal//title`, the size of the input sequence containing *journal* elements varies from around 2,000 to 200,000 elements, whereas the size of the *title* sequence remains stable (roughly 200,000 elements). Fig. 8a illustrates the runtime performance of the *DescFullHashA* operator and the *DescFullHashB* operator for the same query. For selectivities smaller than 10%, the runtime of each operator remains quite the same, because in these cases external memory access costs for the node reference indexes (column sockets) dominate the execution time, whereas the time for the hash table creation and probing remains roughly the same. However for selectivities $>$ 10%, the runtime increases due to higher CPU costs for hashing and probing of larger input sequences. The gap between the `DescFullHashA` and the `DescFullHashB` operator results from hashing the wrong—i. e., the larger—input sequence (*title*) instead of the smaller one (in operator `DescFullHashB`). Therefore, it is important that the query optimizer chooses the right operator for an anticipated data distribution.

4.2 Hash-Based vs. Sort-Based Schemes

In the next test, we want to identify the performance differences of our hash-based schemes as compared to sort-based schemes. For this purpose, we implemented the *StackTree* algorithm [1] and the structural join strategy from [14] called *AxisSort** in the following. Both operators work in two phases: In phase 1, input sequences are sorted using the *QuickSort* algorithm. While *StackTree* needs to sort both input sequences,



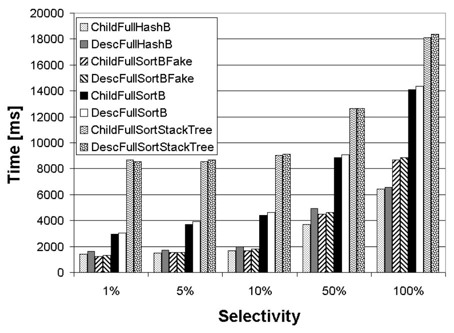


Fig. 8. a) DescFullHash* Characteristics, b) Operator Comparison

*AxisSort** only needs to sort the smaller one. In phase 2, *StackTree* accomplishes its ordinary join strategy, while *AxisSort** performs a binary search on the sorted input for each element of the other input sequence. To compare our operators with minimal-cost sort-based schemes, we introduce hypothetical operators which also sort the smaller input sequence, but omit the probing phase. Thus, so-called **Fake* operators do not produce any output tuples. The result comparison is presented in Fig. 8b. Having the same join selectivity dependency, our hash-based operators are approximately twice as fast as the sort-based operators (with result construction). The figures for the *StackTree* algorithm impressively demonstrate that sort operations on intermediate results in query plans should really be avoided if possible. Finally, the hash-based operators—with their "handicap" to produce a result—match the sort-based fake operators.

4.3 Memory Consumption

Finally, we measured the memory consumption of hash-based and sort-based operators. On the generated document collection, we issued the query `//organization[.//andorra]`, where the number of andorra elements varied from 2000 to 200.000, whereas organization elements remained stable (at roughly 200.000). For comparison, we used the `DescFullHashB`[5] and the `DescFullSortB` operator. In all selectivity ranges, the internal hash table of the hash-based operator consumed three to four times more memory than the plain array of the sort-based one. To reduce this gap, a space optimization for hash-based operators is possible: Each key contained in the hash-table (as depicted in Fig. 6a) is repeated (as a prefix) in the join field value of the tuples contained in the key's queue. This redundant information can safely be disposed for a more compact hash table.

In a last experiment, we compare `DescFullHashB` with `AncHashB`. Here, the semi-join alternative required around three times fewer memory than the full join variant on all selectivities. This circumstance is also a strong argument for our proposal, that the query optimizer should pick semi-join operators whenever possible.

[5] Note, regarding the space complexity, `DescFullHashB` is one of the more expensive representative among the hash-based operators (see 3.2).

5 Conclusions

In this paper, we have considered the improvement of twig pattern queries—a key requirement for XML query evaluation. For this purpose, we have substantially extended the work on structural join algorithms thereby focussing on hashing support. While processing twig patterns, our algorithms, supported by appropriate document store and index structures, primarily rely on SPLIDs which flexibly enable and improve path processing steps by introducing several new degrees of freedom when designing physical operators for path processing steps.

Performance measurements approved our expectations about hash-based operators. They are, in the selectivity range 1%–100%, twice as fast as sort-based schemes and not slower than the **Fake* operators. As another beneficial aspect, intermediate sorts in QEPs can be drastically reduced. Such hash-based operators should be provided—possibly with other kinds of index-based join operators—in a tool box for the cost-based query optimizer to provide for the best QEP generation in all situations.

References

1. S. Al-Khalifa et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141-152 (2002)
2. T. Böhme, E. Rahm: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd DIWeb Workshop: 70-81 (2004)
3. N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: optimal XML pattern matching. Proc. SIGMOD: 310-321 (2002)
4. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. Proc. VLDB: 263-274 (2002)
5. M. Dewey: Dewey Decimal Classification System. http://www.mtsu.edu/ vvesper/dewey.html
6. M. Fernandez, J. Hidders, P. Michiels, J. Simeon, R. Vercammen: Optimizing Sorting and Duplicate Elimination. Proc DEXA: 554-563 (2005).
7. M. Fontoura, V. Josifovski, E. Shekita, B. Yang: Optimizing Cursor Movement in Holistic Twig Joins, Proc. 14th CIKM: 784-791 (2005)
8. G. Gottlob, C. Koch, R. Pichler: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. 30(2): 444-491 (2005)
9. T. Härder, M. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered, accepted for Data & Knowledge Engineering (2006)
10. Q. Li, B. Moon: Indexing and Querying XML Data for Regular Path Expressions. Proc. VLDB: 361-370 (2001)
11. Q. Li, B. Moon: Partition Based Path Join Algorithms for XML Data. Proc. DEXA: 160-170 (2003)
12. P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury: ORDPATHs: Insert-Friendly XML Node Labels. Proc. SIGMOD: 903-908 (2004)
13. I. Tatarinov et al.: Storing and Querying Ordered XML Using a Relational Database System. Proc. SIGMOD: 204-215 (2002)
14. Z. Vagena, M. M. Moro, V. J. Tsotras: Efficient Processing of XML Containment Queries using Partition-Based Schemes. Proc. IDEAS: 161-170 (2004)
15. Y. Wu, J. M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. Proc. ICDE: 443-454 (2003).
16. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohmann, On Supporting Containment Queries in Relational Database Management Systems. Proc. SIGMOD: 425-436 (2001)

Efficiently processing XML queries over fragmented repositories with PartiX

Alexandre Andrade[1], Gabriela Ruberg[1], Fernanda Baião[2], Vanessa P. Braganholo[1], and Marta Mattoso[1]

[1] Computer Science Department, COPPE/Federal Univ. of Rio de Janeiro, Brazil
[2] Applied Informatics Department, University of Rio de Janeiro, Brazil
{alexsilv,gruberg,vanessa,marta}@cos.ufrj.br,
fernanda.baiao@uniriotec.br

Abstract. The data volume of XML repositories and the response time of query processing have become critical issues for many applications, especially for those in the Web. An interesting alternative to improve query processing performance consists in reducing the size of XML databases through fragmentation techniques. However, traditional fragmentation definitions do not directly apply to collections of XML documents. This work formalizes the fragmentation definition for collections of XML documents, and shows the performance of query processing over fragmented XML data. Our prototype, PartiX, exploits intra-query parallelism on top of XQuery-enabled sequential DBMS modules. We have analyzed several experimental settings, and our results showed a performance improvement of up to a 72 scale up factor against centralized databases.

1 Introduction

In the relational [18] and object-oriented data models [4], data fragmentation has been used successfully to efficiently process queries. One of the key factors to this success is the formal definition of fragments and their correctness rules for transparent query decomposition. Recently, several fragmentation techniques for XML data have been proposed in literature [1, 2, 6–8, 12]. Each of these techniques aims at a specific scenario: data streams [7], peer-to-peer [1, 6], Web-Service based systems [2], etc.

In our work, we focus on high performance of XML data servers. In this scenario, we may have a single large document (SD), or large collections of documents (MD) over which XML queries are posed. For this scenario, however, existent fragmentation techniques [1, 2, 6–8, 12] do not apply. This is due to several reasons. First of all, they do not clearly distinguish between horizontal, vertical and hybrid fragmentation, which makes it difficult to automatically decompose queries to run over the fragments. Second, none of them present the fragmentation correctness rules, which are essential for the XML data server to verify the correctness of the XML fragments and then apply the reconstruction rule to properly decompose queries. Also, for large XML repositories, it is important to have a fragmentation model close to the traditional fragmentation techniques, so it can profit as much as possible from well-known results. Third, the query processing techniques are specific for the scenarios where they were proposed, and thus do not apply to our scenario. For instance, the model proposed in [7] for stream data does not support horizontal fragmentation. The same happens in [2], where fragmentation is used for efficient XML data exchange through Web services. Finally, the lack of distinction between SD and MD prevents the distributed query processing of the MD collection [6, 12].

Thus, to efficiently answer queries over large XML repositories using an XML data server, we need a precise definition of XML fragmentation and a high performance environment, such as a cluster of PCs. This way, queries can be decomposed in subqueries which may run in parallel at each cluster node, depending on how the database is fragmented. In this paper, we are interested in the empirical assessment of data fragmentation techniques for XML repositories. We formalize the main fragmentation alternatives for collections of XML documents. We also contribute by defining the rules that verify the correctness of a fragment definition. Our fragmentation model is formal and yet simple when compared to related work. We consider both SD and MD repositories. To address the lack of information on the potential gains that can be achieved with partitioned XML repositories, we present experimental results for horizontal, vertical and hybrid fragmentation of collections of XML documents. The experiments were run with our prototype named PartiX. The architecture of PartiX and sketches of algorithms for query decomposition and result composition are available at [3]. Our results show substantial performance improvements, of up to a 72 scale up factor compared to the centralized setting, in some relevant scenarios.

This paper is organized as follows. In Section 2, we discuss related work. Section 3 presents some basic concepts on XML data model and query language, and formalizes our fragmentation model. Our experimental results and corresponding analysis are presented in Section 4. Section 5 closes this work with some final remarks and research perspectives.

2 Related Work

In this section, we briefly present related work. A more detailed discussion can be found in [3]. Foundations of distributed database design for XML were first addressed in [8] and [12]. Ma and Schewe [12] propose three types of XML fragmentation: *horizontal*, which groups elements of a single XML document according to some selection criteria; *vertical*, to restructure a document by unnesting some elements; and a special type named *split*, to break an XML document into a set of new documents. However, these fragmentation types are not clearly distinguished. For example, horizontal fragmentation involves data restructuring and elements projection, thus yielding fragments with different schema definitions.

Our definition of vertical XML fragmentation is inspired in the work of Bremer and Gertz [8]. They propose an approach for distributed XML design, covering both data fragmentation and allocation. Nevertheless, their approach only addresses SD repositories. Moreover, their formalism does not distinguish between horizontal and vertical fragmentation, which are combined in a hybrid type of fragment definition. They maximize local query evaluation by replicating global information, and distributing some indexes. They present performance improvements, but their evaluation focuses on the benefits of such indexes.

Different definitions of XML fragments have been used in query processing over streamed data [7], peer-to-peer environments [1, 6], and Web-Service based scenarios [2]. However, they either do not present fragmentation alternatives to SD and MD [6], or do not distinguish between the different fragmentation types [1, 2, 6, 7]. In PartiX, we support horizontal, vertical and hybrid fragmentation of XML data for SD and MD repositories. Furthermore, we have implemented a PartiX prototype, and performed several tests to evaluate the performance of these fragmentation alternatives. No work in the literature presents experimental analysis of the query processing response time on fragmented XML repositories.

3 XML Data Fragmentation

3.1 Basic Concepts

XML documents consist of trees with nodes labeled by element names, attribute names or constant values. Let $\mathscr{L}$ be the set of distinct element names, $\mathscr{A}$ the set of distinct attribute names, and $\mathscr{D}$ the set of distinct data values. An *XML data tree* is denoted by the expression $\Delta := \langle t, \ell, \Psi \rangle$, where: t is a finite ordered tree, ℓ is a function that labels nodes in t with symbols in $\mathscr{L} \cup \mathscr{A}$; and Ψ maps leaf nodes in t to values in $\mathscr{D}$. The root node of Δ is denoted by $root_\Delta$. We assume nodes in Δ do not have mixed content; if a given node v is mapped into $\mathscr{D}$, then v does not have siblings in Δ. Notice, however, that this is not a limitation, but rather a presentation simplification. Furthermore, nodes with labels in $\mathscr{A}$ have a single child whose label must be in $\mathscr{D}$. An XML document is a data tree.

Basically, names of XML elements correspond to names of data types, described in a DTD or XML Schema. Let S be a schema. We say that document $\Delta := \langle t, \ell, \Psi \rangle$ *satisfies* a type τ, where $\tau \in S$, iff $\langle t, \ell \rangle$ is a tree derived from the grammar defined by S such that $\ell(root_\Delta) \rightarrow \tau$. A *collection* C of XML documents is a set of data trees. We say it is *homogeneous* if all the documents in C satisfy the same XML type. If not, we say the collection is *heterogeneous*. Given a schema S, a homogeneous collection C is denoted by the expression $C := \langle S, \tau_{root} \rangle$, where τ_{root} is a type in S and all instances Δ of C satisfy τ_{root}.

Figure 1(a) shows the $S_{virtual\_store}$ schema tree, which we use in the examples throughout the paper. In this Figure, we indicate the minimum and maximum cardinalities (assuming cardinality 1..1 when omitted). The main types in $S_{virtual\_store}$ are `Store` and `Item`, which describe a virtual store and the items it sells. Items are associated with sections and may have descriptive characteristics. Items may also have a list of pictures to be used in the virtual store, and a history of prices. Figure 1(b) shows the definition of the homogeneous collections C_{store} and C_{items}, based on $S_{virtual\_store}$.

We consider two types of XML repositories, as mentioned in [17]. An XML repository may be composed of several documents (*Multiple Documents*, MD) or by a single large document which contains all the information needed (*Single Document*, SD). The collection C_{items} of Figure 1(b) corresponds to an MD repository, whereas the collection C_{store} is an SD repository.

A *path expression* P is a sequence $/\mathsf{e}_1/\ldots/\{\mathsf{e}_k \mid @\mathsf{a}_k\}$, where $\mathsf{e}_x \in \mathscr{L}$, $1 \leq x \leq k$, and $\mathsf{a}_k \in \mathscr{A}$. P may optionally contain the symbols "$*$" to indicate any element, and "//"

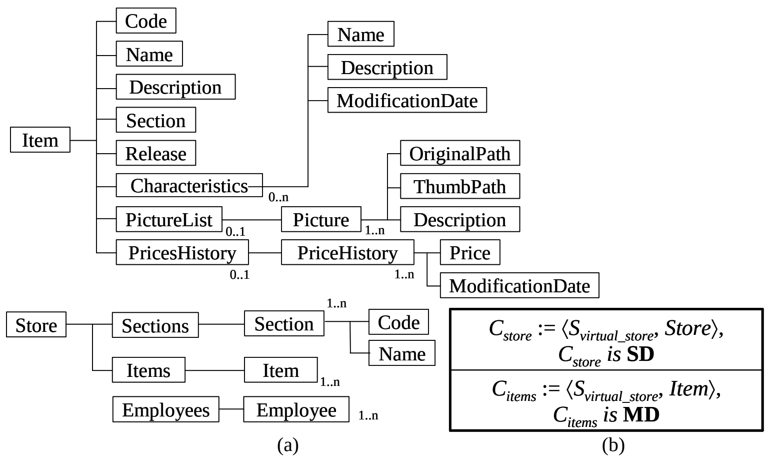


Fig. 1. (a) $S_{virtual\_store}$ schema (b)Specification of collections C_{Store} and C_{Items}

to indicate any sequence of descendant elements. Besides, the term $\mathsf{e}[i]$ may be used to denote the i-th occurrence of element e. The evaluation of a path expression P in a document Δ represents the selection of all nodes with label e_k (or a_k) whose steps from $root_\Delta$ satisfy P. P is said to be *terminal* if the content of the selected nodes is simple (that is, if they have domain in $\mathscr{D}$). On the other hand, a *simple predicate* p is a logical expression: $p := \ P\ \theta\ value\ \mid\ \phi_v(P)\ \theta\ value\ \mid\ \phi_b(P)\ \mid\ Q$, where P is a terminal path expression, $\theta \in \{=,<,>,\neq,\leq,\geq\}$, $value \in \mathscr{D}$, ϕ_v is a function that returns values in $\mathscr{D}$, ϕ_b is a boolean function and Q denotes an arbitrary path expression. In the latter case, p is true if there are nodes selected by Q (existential test).

3.2 XML Fragmentation Techniques

The subject of data fragmentation is well known in relational [15] and object databases [4]. Traditionally, we can have three types of fragments: *horizontal*, where instances are grouped by selection predicates; *vertical*, which “cuts” the data structure through projections; and/or *hybrid*, which combines selection and projection operations in its definition. Our XML fragmentation definition follows the semantics of the operators from the TLC algebra [16], since it is one of the few XML algebras [9, 10, 18] that uses collections of documents, and thus is adequate to the XML data model defined in Section 3.1. In [3], we show how fragment definitions in PartiX can be expressed with TLC operators. In XML repositories, we consider that the fragmentation is defined over the schema of an XML collection. In the case of an MD XML database, we assume that the fragmentation can only be applied to homogeneous collections.

Definition 1. *A* fragment *F of a homogeneous collection C is a collection represented by $F := \langle C, \gamma\rangle$, where γ denotes an operation defined over C. F is* horizontal *if γ denotes a selection;* vertical*, if operator γ is a projection; or* hybrid*, when there is a composition of operators select and project.*

(a)	$F1_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section="CD"}} \rangle$ $F2_{CD} := \langle C_{items}, \sigma_{\text{/Item/Section}\neq\text{"CD"}} \rangle$	(b)	$F1_{good} := \langle C_{items}, \sigma_{\text{contains(//Desciption, "good")}} \rangle$ $F2_{good} := \langle C_{items}, \sigma_{\text{not(contains(//Desciption, "good"))}} \rangle$
		(c)	$F1_{with\_pictures} := \langle C_{items}, \sigma_{\text{/Item/PictureList}} \rangle$ $F2_{with\_pictures} := \langle C_{items}, \sigma_{\text{empty(/Item/PictureList)}} \rangle$

Fig. 2. Examples of three alternative fragments definitions over the collection C_{items}

Instances of a fragment F are obtained by applying γ to each document in C. The set of the resulting documents form the fragment F, which is valid if all documents generated by γ are well-formed (i.e., they must have a single root).

We now detail and analyze the main types of fragmentation in XML. However, we first want to make clear our goal in this paper. Our goal is to show the advantages of fragmenting XML repositories in query processing. Therefore, we formally define the three typical types of XML fragmentation, present correctness criteria for each of them, and compare the performance of queries stated over fragmented databases with queries over centralized databases.

Horizontal Fragmentation. This technique aims to group data that is frequently accessed in isolation by queries with a given selection predicate. A horizontal fragment F of a collection C is defined by the selection operator (σ) [10] applied over documents in C, where the predicate of σ is a boolean expression with one or more simple predicates. Thus, F has the same schema of C.

Definition 2. *Let μ be a conjunction of simple predicates over a collection C. The horizontal fragment of C defined by μ is given by the expression $F := \langle C, \sigma_\mu \rangle$, where σ_μ denotes the selection of documents in C that satisfy μ, that is, F contains documents of C for which σ_μ is true.*

Figure 2 shows the specification of some alternative horizontal fragments for the collection C_{items} of Figure 1(b). For instance, fragment $F1_{good}$ (Figure 2(b)) groups documents from C_{items} which have `Description` nodes that satisfy the path expression `//Description` (that is, `Description` may be at any level in C_{items}) and that contain the word "good". Alternatively, one can be interested in separating, in different fragments, documents that have/have not a given structure. This can be done by using an existential test, and it is shown in Figure 2(c). Although $F1_{with\_pictures}$ and C_{items} have the same schema, in practice they can have different structures, since the element used in the existential test is mandatory in $F1_{with\_pictures}$. Observe that $F1_{with\_pictures}$ cannot be classified as a vertical nor hybrid fragment.

Notice that, by definition, SD repositories may not be horizontally fragmented, since horizontal fragmentation is defined over documents (instead of nodes). However, the elements in an SD repository may be distributed over fragments using a hybrid fragmentation, as described later in this paper.

Vertical Fragmentation. It is obtained by applying the projection operator (π) [16] to "split" a data structure into smaller parts that are frequently accessed in queries. Observe that, in XML repositories, the projection operator has a quite sophisticated semantics: it is possible to specify projections that exclude subtrees whose root is located in any level of an XML tree. A projection over a collection C retrieves, in each document of

(a)	$F1_{items} := \langle C_{items}, \pi_{\text{/Item, \{/Item/PictureList\}}} \rangle$ $F2_{items} := \langle C_{items}, \pi_{\text{/Item/PictureList, \{\}}} \rangle$	(b)	$F1_{sections} := \langle C_{store}, \pi_{\text{/Store/Sections, \{\}}} \rangle$ $F2_{section} := \langle C_{store}, \pi_{\text{/Store, \{/Store/Sections\}}} \rangle$

Fig. 3. Examples of vertical fragments definitions over collections C_{items} and C_{store}

$$F1_{items} := \langle C_{store}, \pi_{\text{/Store/Items, \{\}}} \bullet \sigma_{\text{/Item/Section="CD"}} \rangle$$
$$F2_{items} := \langle C_{store}, \pi_{\text{/Store/Items, \{\}}} \bullet \sigma_{\text{/Item/Section="DVD"}} \rangle$$
$$F3_{items} := \langle C_{store}, \pi_{\text{/Store/Items,\{\}}} \bullet \sigma_{\text{/Item/Section}\neq\text{"CD"}\wedge\text{ /Item/Section}\neq\text{"DVD"}} \rangle$$
$$F4_{items} := \langle C_{store}, \pi_{\text{/Store,\{/Store/Items\}}} \rangle$$

Fig. 4. Examples of hybrid fragments over collection C_{store}

C (notice that C may have a single document, in case it is of type SD), a set of subtrees represented by a path expression, which are possibly pruned in some descendant nodes.

Definition 3. *Let P be a path expression over collection C. Let $\Gamma := \{E_1, \ldots, E_x\}$ be a (possibly empty) set of path expressions contained in P (that is, path expressions in which P is a prefix). A* vertical fragment *of C defined by P is denoted $F := \langle C, \pi_{P,\Gamma} \rangle$, where $\pi_{P,\Gamma}$ denotes the projection of the subtrees rooted by nodes selected by P, excluding from the result the nodes selected by the expressions in Γ. The set Γ is called the* prune criterion *of F.*

It is worth mentioning that the path expression P cannot retrieve nodes that may have cardinality greater than one (as it is the case of `/Item/PictureList/Picture`, in Figure 1(a)), except when the element order is indicated (e.g. `/Item/PictureList/Picture[1]`). This restriction assures that the fragmentation results in well-formed documents, without the need of generating artificial elements to reorganize the subtrees projected in a fragment.

Figure 3 shows examples of vertical fragments of the collections C_{items} and C_{store}, defined on Figure 1(b). Fragment $F2_{items}$ represents documents that contain all `PictureList` nodes that satisfy the path `/Item/PictureList` in the collection C_{items} (no prune criterion is used). On the other hand, the nodes that satisfy `/Item/PictureList` are exactly the ones pruned out the subtrees rooted in `/Item` in the fragment $F1_{items}$, thus preserving disjointness w.r.t. $F2_{items}$.

Hybrid Fragmentation. The idea here is to apply a vertical fragmentation followed by a horizontal fragmentation, or vice-versa. An interesting use of this technique is to normalize the schema of XML collections in SD repositories, thereby allowing horizontal fragmentation.

Definition 4. *Let σ_μ and $\pi_{P,\Gamma}$ be selection and projection operators, respectively, defined over a collection C. A* hybrid fragment *of C is represented by $F := \langle C, \pi_{P,\Gamma} \bullet \sigma_\mu \rangle$, where $\pi_{P,\Gamma} \bullet \sigma_\mu$ denotes the selection of the subtrees projected by $\pi_{P,\Gamma}$ that satisfy σ_μ.*

The order of the application of the operations in $\pi_{P,\Gamma} \bullet \sigma_\mu$ depends on the fragmentation design. Examples of hybrid fragmentation are shown in Figure 4.

3.3 Correctness Rules of the Fragmentation

An XML distribution design consists of fragmenting collections of documents (SD or MD) and allocating the resulting fragments in sites of a distributed system, where each collection is associated to a set of fragments. Consider that a collection C is decomposed into a set of fragments $\Phi := \{F_1, \ldots, F_n\}$. The following rules must be verified to guarantee the fragmentation of C is correct:

– *Completeness*: each data item in C must appear in at least one fragment $F_i \in \Phi$. In the horizontal fragmentation, the data item consists of an XML document, while in the vertical fragmentation, it is a node.
– *Disjointness*: for each data item d in C, if $d \in F_i$, $F_i \in \Phi$, then d cannot be in any other fragment $F_j \in \Phi$, $j \neq i$.
– *Reconstruction*: it must be possible to define an operator ∇ such that $C := \nabla F_i, \forall F_i \in \Phi$, where ∇ depends on the type of fragmentation. For horizontal fragmentation, the union ($\cup$) operator [10] is used (TLC is an extension of TAX [10]), and for vertical fragmentation, the join ($\bowtie$) operator [16] is used. We keep an ID in each vertical fragment for reconstruction purposes.

These rules are important to guarantee that queries are correctly translated from the centralized environment to the fragmented one, and that results are correctly reconstructed. Procedures to verify correctness depend on the algorithms adopted in the fragmentation design. As an example, some fragmentation algorithms for relations guarantee the correctness of the resulting fragmentation design [15]. Still others [14] require use of additional techniques to check for correctness. Such automatic verification is out of the scope of this paper.

Query Processing. By using our fragmentation definition, we can adopt a query processing methodology similar to the relational model [15]. The query expressed on the global XML documents can be mapped to its corresponding fragmented XML documents by using the fragmentation schema definition and reconstruction program. Then an analysis on query specification versus schema definition can proceed with data localization. Finally global query optimization techniques can be adopted [10, 16].

4 Experimental Evaluation

This section presents experimental results obtained with the PartiX implementation for horizontal, vertical and hybrid fragmentation. We evaluate the benefits of data fragmentation for the performance of query processing in XML databases. We used a 2.4Ghz Athlon XP with 512Mb RAM memory in our tests. We describe the experimental scenario we have used for each of the fragmentation types: horizontal, vertical and hybrid, and show that applying them in XML data collections have reduced the query processing times.

We applied the ToXgene [5] XML database generator to create the C_{store} and C_{items} collections, as defined in Figures 1(a) and (b), and also a collection for the schema defined in the XBench benchmark [17]. All of them were stored in the eXist DBMS [13]. Four databases were generated for the tests: *database ItemsSHor*, with document sizes of 2K in average, and elements `PriceHistory` and `ImagesList` with zero occurrences (Figure 1(a)); *database ItemsLHor*, with document sizes of 80Kb in average (Figure 1(a)); *database XBenchVer*, with the XBench collections, with document sizes varying from 5Mb to 15Mb each; and database StoreHyb (Figure 1(a)), with document sizes from 5Mb to 500Mb. Experiments were conducted varying the number of documents in each database to evaluate the performance of fragmentation for different database sizes (5Mb, 20Mb, 100Mb and 250Mb for all databases, and 500Mb for databases ItemsLHor and StoreHyb). (Due to space restrictions, in this paper we show only the results for the 250Mb database. The remaining results are available at [3].) Some indexes were automatically created by the eXist DBMS to speed up text search operations and path expressions evaluation. No other indexes were created.

Each query was decomposed in sub-queries (according to [3]) to be processed with specific data fragments. When the query predicates match the fragmentation predicates, the sub-queries are issued only to the corresponding fragments. After each sub-query is executed, we compose the result to compute the final query answer [3]. The parallel execution of a query was simulated assuming that all fragments are placed at different sites and that the sub-queries are executed in parallel in each site. For instance, in Figure 5(a) with 8 fragments, we can have at most 8 sub-queries running in parallel. We have used the time spent by the slowest site to produce the result. We measured the communication time for sending the sub-queries to the sites and for transmitting their partial results, since there is no inter-node communication. This was done by calculating the average size of the result and dividing it by the Gigabit Ethernet transmission speed. For all queries we have measured the time between the moment PartiX receives the query until final result composition.

In our experiments, each query was submitted 10 times, and the execution time was calculated by discarding the first execution time and calculating the average of the remaining results. We have measured the execution times of each sub-query. More details on our experiments are available in [3].

Horizontal Fragmentation. For horizontal fragmentation, the tests were run using a set of 8 queries [3], which illustrates diverse access patterns to XML collections, including the usage of predicates, text searches and aggregation operations. The XML database was fragmented as follows. The C_{Items} collection was horizontally fragmented by the "Section" element, following the correctness rules of Section 3.3. We varied the number of fragments (2, 4 and 8) with a non-uniform document distribution. The fragments definitions are shown in [3].

Figure 5(a) contains the performance results of the PartiX execution on top of database ItemsSHor, and Figure 5(b) on database ItemsLHor, in the scenarios previously described. The results show that the fragmentation reduces the response time for most of the queries. When comparing the results of databases ItemsSHor and ItemsLHor with a large number of documents, we observe that the eXist DBMS presents better results when dealing with large documents. This is due to some pre-processing operations (e.g., parsing) that are carried out for each XML tree. For example, when using a 250Mb database size and centralized databases, query Q8 is executed in 1200s in ItemsSHor, and in 31s in ItemsLHor. When using 2 fragments, these times are reduced to 300s and 14s, respectively. Notice this is a superlinear speedup. This is very common also in relational databases, and is due to reduction of I/O operations and better use of machine resources such as cache and memory, since a smaller portion of data is being processed at each site.

An important conclusion obtained from the experiments relates to the benefit of horizontal fragmentation. The execution time of queries with text searches and aggre-

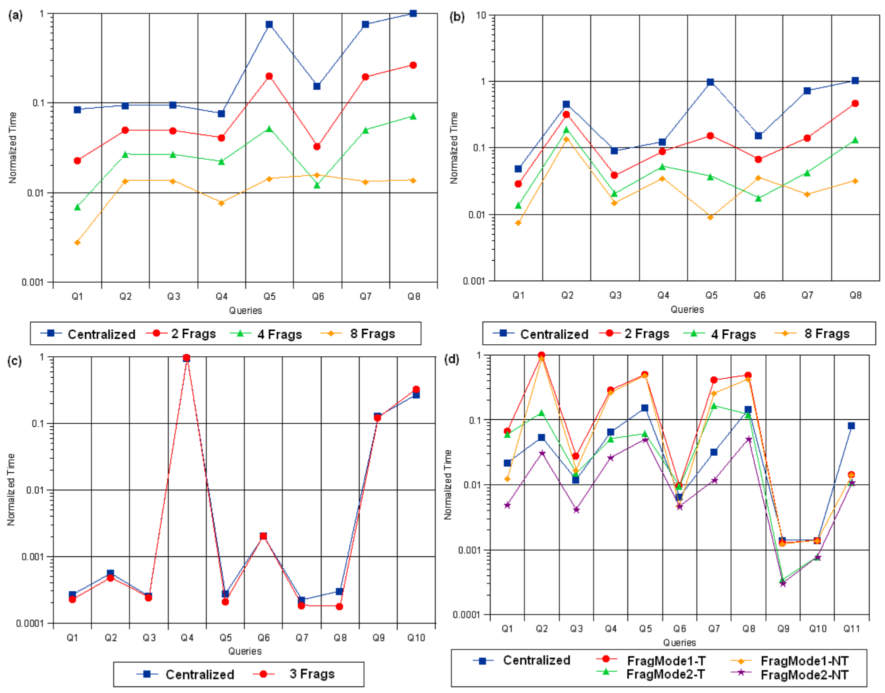


Fig. 5. Experimental results for databases (a) ItemsSHor and (b) ItemsLHor - horizontal fragmentation; (c) XBenchVer - vertical fragmentation; (d) StoreHyb with and without transmission times - hybrid fragmentation

gation operations (Q5, Q6, Q7 and Q8) is significantly reduced when the database is horizontally fragmented. It is worth mentioning that text searches are very common in XML applications, and typically present poor performance in centralized environments, sometimes prohibiting their execution. This problem also occurs with aggregation operations. It is important to notice that our tests included an aggregation function (count) that may be entirely evaluated in parallel, not requiring additional time for reconstructing the global result.

Another interesting result can be seen in the execution of Q6. As the number of fragments increases, the execution time of Q6 increases in some cases. This happens because eXist generates different execution plans for each sub-query, thus favoring the query performance in case of few fragments. Yet, all the distributed configurations performed better than the centralized database.

As expected, in small databases (i.e., 5Mb) the performance gain obtained is not enough to justify the use of fragmentation [3]. Moreover, we concluded that the document size is very important for defining the fragmentation schema. Database ItemsLHor (Figure 5(b)) presents better results with few fragments, while database ItemsSHor presents better results with many fragments.

Vertical Fragmentation. For the experiments with vertical fragmentation, we have used the XBenchVer database and some of the queries specified in XBench [17], which are shown in [3]. We have named them Q1 to Q10, although these names do not correspond to the names used in the XBench document.

Database *XBenchVer* was vertically fragmented in three fragments: $F1_{papers} := \langle C_{papers}, \pi_{/article/prolog} \rangle$, $F2_{papers} := \langle C_{atigos}, \pi_{/article/body} \rangle$, and $F3_{papers} := \langle C_{artigos}, \pi_{/article/epilog} \rangle$. Figure 5(c) shows the performance results of PartiX in this scenario. In the 5Mb database, we had gains in all queries, except for Q4 and Q10 [3]. With vertical fragmentation, the main benefits occur for queries that use a single fragment. Since queries Q4, Q7, Q8 and Q9 need more than one fragment, they can be slowed down by fragmentation. Query Q4 does not present performance gains in any case, except for a minor improvement in the 100Mb database [3]. We believe once more that some statistics or query execution plan has favored the execution of Q4 in this database. In the 20Mb database, all queries presented performance gains (except for Q4), including Q10, which had presented poor performance in the 5Mb database.

As the database size grows, the performance gains decreases. In the 250Mb database, queries Q6, Q9 and Q3 perform equivalently to the centralized approach. With these results, we can see that vertical fragmentation is useful when the queries use few fragments. The queries with bad performance were those involving text search, since in general, they must be applied to all fragments. In such case, the performance is worse than for horizontal fragmentation, since the result reconstruction requires a join (much more expensive than a union).

Hybrid Fragmentation. In the experiments with hybrid fragmentation, we have used the C_{Store} collection fragmented into 5 different fragments. Fragment F_1 prunes */Store/Items*, while the remaining 4 fragments are all about Items, each of them horizontally fragmented over */Store/Items/Item/Section*. We call this database *StoreHyb*, and the set of queries defined over it is shown in [3].

As we will see later on, the experimental results with hybrid fragmentation were heavily influenced by the size of the returning documents. Because of this, we show the performance results with and without the transmission times.

We consider the same queries and selection criteria adopted for databases ItemsSHor e ItemsLHor, with some modifications. With this, most of the queries returned all the content of the “Item” element. This was the main performance problem we have encountered, and it affected all queries. This serves to demonstrate that, besides a good fragmentation design, queries must be carefully specified, so that they do not return unnecessary data. Because XML is a verbose format, an unnecessary element may carry a subtree of significant size, and this will certainly impact in the query execution time.

Another general feature we have noticed while making our tests was that the implementation of the horizontal fragment affects the performance results of the hybrid fragmentation. To us, it was natural to take the single document representing the collection, use the prune operation, and, for each “Item” node selected, to generate an independent document and store it. This approach, which we call *FragMode1*, has proved to be very inefficient. The main reason for this is that, in these cases, the query processor has to parse hundreds of small documents (the ones corresponding to the “Item” fragments), which is slower than parsing a huge document a single time. To solve this problem, we have implemented the horizontal fragmentation with a single document (SD), exactly like the original document, but with only the item elements obtained by the selection operator. We have called this approach *FragMode2*. As we will see, this fragmentation mode beats the centralized approach in most of the cases.

When we consider the transmission times (*FragModeX-T* in Figure 5(e)), FragMode1 performs worse for all database sizes, for all queries, except for queries Q9, Q10 and Q11. Queries Q9 and Q10 are those that prune the "Items" element, which makes the parsing of the document more efficient. Query Q11 uses an aggregation function that presented a good performance in the 100Mb database and in larger ones. In the remaining databases, it presented poor performance (5Mb database) or an anomalous behavior (20Mb database) [3].

Notice the FragMode2 performs better, although it does not beat the centralized approach in all cases. In the 5Mb database, it wins in queries Q3, Q4, Q5 and Q6, which benefit from the parallelism of the fragments and from the use of a specific fragment. As in the FragMode1, queries Q9 and Q10 always performs better than the centralized case; query Q11 only looses in the 5Mb database.

As the database size grows, the query results also increase, thus increasing the total query processing time. In the 20Mb database, query Q6 performs equivalently to the centralized approach. In the 100Mb database, this also happens to Q3 and Q6. In the 250Mb database, these two queries perform worse than in the centralized approach. Finally, in the 500Mb database, query Q4 also performs equivalently to the centralized case, and the remaining ones loose.

As we could see, the transmission times were decisive in the obtained results. Without considering this time, FragMode2 wins in all databases, in all queries, except for query Q11 in the 5Mb database. However, FragMode1 has shown to be effective in some cases. Figure 5(e) shows the experimental results without the transmission times (*FragModeX-NT*). It shows that hybrid fragmentation reduces the query processing times significantly.

5 Conclusions

This work presents a solution to improve the performance in the execution of XQuery queries over XML repositories. This is achieved through the fragmentation of XML databases. We present a formal definition for the different types of XML fragments and define correctness criteria for the fragmentation. By specifying the concept of collections, we create an abstraction where fragment definitions apply to both single and multiple document repositories (SD and MD). These concepts are not found in related work, and they are fundamental to perform query decomposition and result composition.

Our experiments highlight the potential to significant gains of performance through fragmentation. The reduction in the time of query execution is obtained by intra-query parallelism, and also by the local execution, avoiding scanning unnecessary fragments. The PartiX architecture [3] is generic, and can be plugged to any XML DBMS that process XQuery queries. This architecture follows the approach of *database clusters* that have been presenting excellent performance results over relational DBMSs [11].

As future work, we intend to use the proposed fragmentation model to define a methodology for fragmenting XML databases. This methodology could be used define algorithms for the fragmentation design [18], and to implement tools to automate this fragmentation process. We are also working on detailing algorithms to automatically rewrite queries to run over the fragmented database.

Acnowledgements. The authors thank CNPq for partially funding this work. V. Braganholo thanks FAPERJ and G. Ruberg thanks the Central Bank of Brazil for their support. The contents of this paper express the viewpoint of the authors only.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.
2. S. Amer-Yahia and Y. Kotidis. A web-services architecture for efficient XML data exchange. In *ICDE*, pages 523–534, 2004.
3. A. Andrade, G. Ruberg, F. Baião, V. Braganholo, and M. Mattoso. Partix: Processing XQueries over fragmented XML repositories. Technical Report ES-691, COPPE/UFRJ, 2005. `http://www.cos.ufrj.br/~vanessa`.
4. F. Baião, M. Mattoso, and G. Zaverucha. A distribution design methodology for object DBMS. *Distributed and Parallel Databases*, 16(1):45–90, 2004.
5. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *WebDB*, pages 621–632, 2002.
6. A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *WIDM*, pages 48–55, 2004.
7. S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. XPath lookup queries in p2p networks. In *WIDM*, pages 48–55, 2004.
8. J.-M. Bremer and M. Gertz. On distributing XML repositories. In *WebDB*, 2003.
9. M. Fernández, J. Siméon, and P. Wadler. An algebra for XML query. In *FSTTCS*, pages 11–45, 2000.
10. H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *DBPL*, pages 149–164, 2001.
11. A. Lima, M. Mattoso, and P. Valduriez. Adaptive virtual partitioning for olap query processing in a database cluster. In *SBBD*, pages 92–105, 2004.
12. H. Ma and K.-D. Schewe. Fragmentation of XML documents. In *SBBD*, 2003.
13. W. Meier. eXist: Open source native XML database, 2000. Available at: `http://exist.sourceforge.net`.
14. S. Navathe, K. Karlapalem, and M. Ra. A mixed fragmentation methodology for initial distributed database design. *Journal of Computer and Software Engineering*, 3(4), 1995.
15. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
16. S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, 2004.
17. B. Yao, M. Ozsu, and N. Khandelwal. Xbench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–632, 2004.
18. X. Zhang, B. Pielech, and E. Rundesnteiner. Honey, I shrunk the XQuery!: an XML algebra optimization approach. In *WIDM*, pages 15–22, 2002.

Validity-Sensitive Querying of XML Databases

Slawomir Staworko and Jan Chomicki

University at Buffalo,
{staworko,chomicki}@cse.buffalo.edu

Abstract. We consider the problem of querying XML documents which are not valid with respect to given DTDs. We propose a framework for measuring the invalidity of XML documents and compactly representing minimal repairing scenarios. Furthermore, we present a validity-sensitive method of querying XML documents, which extracts more information from invalid XML documents than does the standard query evaluation. Finally, we provide experimental results which validate our approach.

1 Introduction

XML is rapidly becoming the standard format for the representation and exchange of semi-structured data (documents) over the Internet. In most contexts, documents are processed in the presence of a schema, typically a Document Type Definition (DTD) or an XML Schema. Although currently there exist various methods for maintaining the validity of semi-structured databases, many XML-based applications operate on data which is invalid with respect to a given schema. A document may be the result of integrating several documents of which some are not valid. Parts of an XML document could be imported from a document that is valid with respect to a schema slightly different than the given one. For example, the schemas may differ with respect to the constraints on the cardinalities of elements. The presence of legacy data sources may even result in situations where schema constraints cannot be enforced at all. Also, temporary violations of the schema may arise during the updating of an XML database in an incremental fashion or during data entry.

At the same time, DTDs and XML Schemas capture important information about the expected structure of an XML document. The way a user formulates queries in an XML query language is directly influenced by her knowledge of the schema. However, if the document is not valid, then the result of query evaluation may be insufficiently informative or may fail to conform to the expectations of the user.

Example 1. Consider the DTD D_0 in Figure 1 specifying a collection of project descriptions: Each project description consists of a name, a manager, a collection of subprojects, and a collection of employees involved in the project. The following query Q_0 computes the salaries of all employees that are not managers:

$$/projs//proj/name/following\_sibling::emp/following\_sibling::emp/salary$$

Now consider the document T_0 in Figure 2 which lacks the information about the manager of the main project. Such a document can be the result of the main project not having the manager assigned yet or the manager being changed.

```
<!ELEMENT projs  (proj*)>
<!ELEMENT proj   (name,emp,proj*,emp*)>
<!ELEMENT emp    (name,salary)>
<!ELEMENT name   (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
```

Fig. 1. DTD D_0

```
<projs><proj>
  <name> Cooking Pierogies </name>
  <proj>
    <name> Preparing Stuffing </name>
    <emp><name> John </name>
         <salary> 80K </salary></emp>
    <emp><name> Mary </name>
         <salary> 40K </salary></emp>
  </proj>
  <emp><name> Peter </name>
       <salary> 30K </salary></emp>
  <emp><name> Steve </name>
       <salary> 50K </salary></emp>
</proj></projs>
```

Fig. 2. An invalid document T_0.

The standard evaluation of the query Q_0 will yield the salaries of *Mary* and *Steve*. However, knowing the DTD D_0, we can determine that an `emp` element following the `name` element `''Cooking Pierogies''` is likely to be missing, and conclude that the salary of *Peter* should also be returned.

Our research addresses the impact of invalidity of XML documents on the result of query evaluation. The problem of querying invalid XML documents has been addressed in the literature in two different ways: through *query modification* or through *document repair*. Query modification involves various techniques of distorting, relaxing, or approximating queries [14, 21, 3]. Document repair involves techniques of cleaning and correcting databases [9, 17]. Our approach follows the second direction, document repair, by adapting the framework of *repairs* and *consistent query answers* developed in the context of inconsistent relational databases [4]. A *repair* is a consistent database instance which is *minimally* different from the original database. Various different notions of minimality have been studied, e.g., set- or cardinality-based minimality. A *consistent query answer* is an answer obtained in *every* repair. The framework of [4] is used as a foundation for most of the work in the area of querying inconsistent databases (for recent developments see [8, 12]).

In our approach, differences between XML documents are captured using sequences of atomic operations on documents: inserting/deleting a leaf. Such operations are used in the context of incremental integrity maintenance of XML documents [1, 5, 6] (modification of a node's label is also handled but we omit it because of space limitations). We define *repairs* to be valid documents obtained from a given invalid document using sequences of atomic operations of *minimum cost*, where the cost is measured simply as the number of operations. Valid answers are defined analogously to consistent answers. We consider schemas of XML documents defined using DTDs.

Example 2. The validity of the document T_1 from Example 1 can be restored in two alternative ways:

1. by inserting in the main project a missing `emp` element (together with its subelements `name` and `salary`, and two text elements). The cost is 5.
2. by deleting the main project node and all its subelements. The cost is 19.

Because of the minimum-cost requirement, only the first way leads to a repair. Therefore, the valid answers to Q_0 consist of the salaries of *Mary*, *Steve*, and *Peter*.

In our opinion, the set of atomic document operations proposed here is sufficient for the correction of *local* violations of validity created by missing or superfluous nodes. The notion of valid query answer provides a way to query possibly invalid XML documents in a *validity-sensitive* way. It is an open question if other sets of operations can be used to effectively query XML documents in a similar fashion.

The contributions of this paper include:

– A framework for validity-sensitive querying of such documents based on measuring the invalidity of XML documents;
– The notion of a *trace graph* which is a compact representation of all repairs of a possibly invalid XML document;
– Efficient algorithms, based on the trace graph, for the computation of valid query answers to a broad class of queries;
– Experimental evaluation of the proposed algorithms.

2 Basic definitions

In our paper we use a model of XML documents and DTDs similar to those commonly used in the literature [5, 6, 16, 19].

Ordered labeled trees We view XML documents as labeled ordered trees with text values. For simplicity we ignore attributes: they can be easily simulated using text values. By Σ we denote a fixed (and finite) set of tree node labels and we distinguish a label $\mathtt{PCDATA} \in \Sigma$ to identify *text nodes*. A text node has no children and is additionally labeled with an element from an infinite domain Γ of text constants. For clarity of presentation, we use capital letters $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \ldots$ for elements from Σ and capital letters $X, Y, Z \ldots$ for variables ranging over Σ.

We assume that the data structure used to store a document allows for any given node to get its label, its parent, its first child, and its following sibling in time $O(1)$. For the purpose of presentation, we represent trees as terms over the signature $\Sigma \setminus \{\mathtt{PCDATA}\}$ with constants from Γ. A text constant $t \in \Gamma$, viewed as a term, represents a tree node with no children, labeled with $\mathtt{PCDATA}$, and whose additional text label is t.

DTDs For simplicity our view of DTDs omits the specification of the root label. A *DTD* is a function D that maps labels from Σ to regular expressions over Σ. We consider only DTDs that for $\mathtt{PCDATA}$ return ε. The size of D, denoted $|D|$, is the sum of lengths of all regular expressions in D.

A tree $T = X(T_1, \ldots, T_n)$ is *valid* w.r.t. a DTD D if: (1) T_i is valid w.r.t. D for every i and, (2) if $X_1, \ldots, X_n$ are labels of root nodes of $T_1, \ldots, T_n$ respectively and $E = D(X)$, then $X_1 \cdots X_n \in L(E)$.

Example 3. Consider the DTD $D_1(\mathtt{A}) = \mathtt{PCDATA} + \varepsilon$, $D_1(\mathtt{B}) = \varepsilon$, $D_1(\mathtt{C}) = (\mathtt{A} \cdot \mathtt{B})^*$. The tree $\mathtt{C}(\mathtt{A}(\mathtt{a}), \mathtt{B}(\mathtt{b}), \mathtt{B}())$ is not valid w.r.t. D_1 but the tree $\mathtt{C}(\mathtt{A}(\mathtt{a}), \mathtt{B}())$ is.

To recognize strings satisfying regular expressions we use the standard notion of *non-deterministic finite automaton* (NDFA) $M = \langle \Sigma, S, q_0, \Delta, F \rangle$, where S is a finite set of *states*, $q_0 \in S$ is a distinguished *starting state*, $F \subseteq S$ is the set of *final states*, and $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*.

2.1 Tree edit distance and repairs

Tree operations A *location* is a sequence of natural numbers defined as follows: ε is the location of the root node, and $v \cdot i$ is the location of i-th child of the node at location v. This notion allows us to identify nodes without fixing a tree.

We consider two *atomic tree operations* (or *operations* for short) commonly used in the context of managing XML document integrity [1, 5, 6]:

1. *Deleting* a leaf at a specified location.
2. *Inserting* a leaf at a specified location. If the tree has already a node at this location, we shift the existing node to the right together with any following siblings.

We note that our approach can be easily extended to handle the operation of *Modifying* the label of a node (omitted here because of space limitations).

The following example shows that the order of applying operations is important and therefore we consider sequences (rather than sets) of operations when transforming documents.

Example 4. Consider the tree $\mathtt{A}(\mathtt{B}(), \mathtt{C}())$. If we first insert $\mathtt{D}$ as a second child of $\mathtt{A}$ and then remove $\mathtt{B}$ we obtain the tree $\mathtt{A}(\mathtt{D}(), \mathtt{C}())$. If, however, we first remove $\mathtt{B}$ and then insert $\mathtt{D}$ as a second child of $\mathtt{A}$ we get $\mathtt{A}(\mathtt{C}(), \mathtt{D}())$.

The *cost* of a sequence of operations is defined to be its length, i.e., the number of operations performed when applying the sequence. Two sequences of operations are *equivalent* on a tree T if their application to T yields the same tree. We observe that some sequences may perform redundant operations, for instance inserting a leaf and then removing it. Because we focus on finding cheapest sequences of operations, we restrict our considerations to redundancy-free sequences (those for which there is no equivalent but cheaper sequence).

Definition 1 (Edit distance). *Given two trees T and S, the* edit distance $dist(T, S)$ *between T and S is the minimum cost of transforming T into S.*

For a DTD D and a (possibly invalid) tree T, a sequence of operations is a sequence *repairing T* w.r.t. D if the document resulting from applying the sequence to T is valid w.r.t. D. We are interested in the cheapest repairing sequences of T.

Definition 2 (Distance to a DTD). *Given a document T and a DTD D, the* distance *$dist(T, D)$ of T to D is the minimum cost of repairing T, i.e.*

$$dist(T, D) = \mathsf{Min}\{dist(T, S) | S \text{ is valid w.r.t } D\}.$$

Repairs The notions of distance introduced above allow us to capture the minimality of change required to repair a document.

Definition 3 (Repair). *Given a document T and a DTD D, a document T' is a* repair *of T w.r.t. D if T' is valid w.r.t. D and $dist(T,T') = dist(T,D)$.*

Note that, if repairing a document involves inserting a text node, the corresponding text label can have infinitely many values, and thus in general there can be infinitely many repairs. However, as shown in the following example, even if the operations are restricted to deletions there can be an exponential number of non-isomorphic repairs of a given document.

Example 5. Suppose we work with documents labeled only with $\Sigma = \{\mathtt{A},\mathtt{B},\mathtt{T},\mathtt{F}\}$ and consider the following DTD: $D(\mathtt{A}) = \mathtt{T}\cdot\mathtt{A} + \mathtt{A}\cdot\mathtt{F} + \mathtt{B}\cdot\mathtt{B}$, $(\mathtt{B}) = \varepsilon$, $D(\mathtt{T}) = \varepsilon$, $D(\mathtt{F}) = \varepsilon$. The tree $\mathtt{A}(\mathtt{T}(),\mathtt{A}(\ldots\mathtt{A}(\mathtt{T}(),\mathtt{A}(\mathtt{B}(),\mathtt{B}()),\mathtt{F}())\ldots),\mathtt{F}())$ consisting of $3n+2$ elements has 2^{n-1} repairs w.r.t. D.

3 Computing the edit distance

In this section we present an efficient algorithm for computing the distance $dist(T,D)$ between a document T and a DTD D. The algorithm works in a bottom-up fashion: we compute the distance between a node and the DTD after finding the distance between the DTD and every child of the node.

3.1 Macro operations

Now, we fix a DTD D and a tree $T = X(T_1,\ldots,T_n)$. The base case, when T is a leaf, is handled by taking $n = 0$. We assume that the values $dist(T_i,D)$ have been computed earlier. We recall that the value $dist(T_i,D)$ is the minimum cost of a sequence of atomic tree operations that transforms T_i into a valid tree. Similarly, the value $dist(T,D)$ corresponds to the cheapest sequence *repairing* T. We model the process of repairing T with 3 *macro operations* applied to the root of T:

1. *Deleting* a subtree rooted at a child.
2. *Repairing* recursively the subtree rooted at a child.
3. *Inserting* as a child a minimum-size valid tree whose root's label is Y for some $Y \in \Sigma$.

Each of these macro operations can be translated to a sequence of atomic tree operations. In the case of a repairing operation there can be an exponential number of possible translations (see Example 5), however, for the purpose of computing $dist(T,D)$ we only need to know their cost. Obviously, the cost of deleting T_i is equal to $|T_i|$ and the cost of repairing T_i is equal to $dist(T_i,D)$ (computed earlier). The cost of inserting a minimal subtree can be found using a simple algorithm (omitted here). A sequence of macro operations is a sequence repairing T if the resulting document is valid. The cost of a sequence of macro operations is the sum of the costs of its elements. A sequence of macro operations is equivalent on T to a sequence of atomic operations if their applications to T yield the same tree. Using the macro operations to repair trees is equivalent to atomic operations.

3.2 Restoration graph

Now, let $X_1,\ldots,X_n$ be the sequence of the labels of roots of $T_1,\ldots,T_n$ respectively. Suppose $E = D(X)$ defines the labels of children of the root and let $M_E = \langle \Sigma, S, q_0, \Delta, F\rangle$ be the NDFA recognizing $L(E)$ such that $|S| = O(|E|)$ [15].

To find an optimal sequence of macro operations repairing T, we construct a directed *restoration graph* U_T. The vertices of U_T are of the form q^i for $q \in S$ and $i \in \{0,\ldots,n\}$. The vertex q^i is referred as the *state q in the i-th column* of U_T and corresponds to the state q being reached by M_E after reading $X_1,\ldots,X_i$ processed earlier with some macro operations. The edges of the restoration graph correspond to the macro operations applied to the children of T:

– $q^{i-1} \xrightarrow{Del} q^i$ corresponds to deleting T_i and such an edge exists for any state $q \in S$ and any $i \in \{1,\ldots,n\}$,
– $q^{i-1} \xrightarrow{Rep} p^i$ corresponds to repairing T_i recursively and such an edge exists only if $\Delta(q, X_i, p)$,
– $q^i \xrightarrow{Ins_Y} p^i$ corresponds to inserting before T_i a minimal subtree labeled with Y and such an edge exists only if $\Delta(q, Y, p)$.

A *repairing path* in U_T is a path from q_0^0 to any accepting state in the last column of U_T.

Lemma 1. *For any sequence of macro operations v, v is a sequence repairing T w.r.t. D iff there exists a repairing path (possibly cyclic) in U_T labeled with the consecutive elements of v.*

If we assign to each edge the cost of the corresponding macro operation, the problem of finding a cheapest repairing sequence of macro operations is reduced to the problem finding a shortest path in U_T.

Theorem 1. *$dist(T,D)$ is equal to the minimum cost of a repairing path in U_T.*

Example 6. Figure 3 illustrates the construction of the restoration graph for the document $\mathtt{C(A(a),B(b),B())}$ and the DTD from Example 3. The automaton $M_{(\mathtt{A}\cdot\mathtt{B})^*}$ consists of two states q_0 and q_1; q_0 is both the starting and the only accepting state; $\Delta = \{(q_0, \mathtt{A}, q_1), (q_1, \mathtt{B}, q_0)\}$. The cheapest repairing paths are indicated with bold lines. In each vertex we additionally put the minimum cost of reaching that vertex from q_0^0. Note that the restoration graph represents 3 different repairs: (1) $\mathtt{C(A(a),B(),A(),B())}$, obtained by inserting $\mathtt{A()}$; (2) $\mathtt{C(A(a),B())}$ obtained by deleting the second child; (3) $\mathtt{C(A(a),B())}$ obtained by repairing the second child (removing the text node $\mathtt{b}$) and removing the third child. We note that although isomorphic, the repairs (2) and (3) are not the same because the nodes labeled with $\mathtt{B}$ are obtained from different nodes in the original tree.

3.3 Trace graph

The *trace graph* U_T^* is the subgraph of U_T consisting of only the cheapest repairing paths. Note that if U_T has cycles, only arcs labeled with inserting macro operations can be involved in them. Since the costs of inserting operations are positive, U_T^* is a directed acyclic graph.

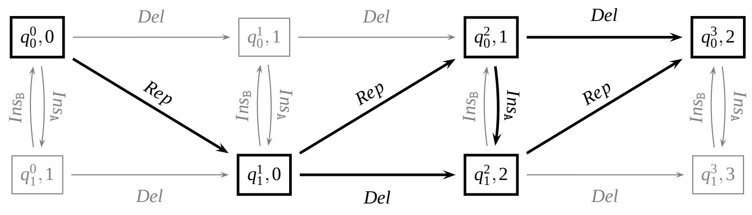


Fig. 3. Construction of the restoration graph.

Repairs and paths in the trace graph Suppose now that we have constructed a trace graph in every node of T. Every repair can be characterized by selecting a path on each of the trace graphs. Similarly a choice of paths in each of the trace graphs yields a repair. We note that a choice of a path on the top-level trace graph of T may correspond to more than one repair (this happens when some subtree has more than one repair).

Complexity analysis We assume that Σ is fixed and $|S|$ is bounded by $|D|$.

Theorem 2. *The distance between a document T and a DTD D can be computed in $O(|D|^2 \times |T|)$ time using $O(|D|^2 \times height(T))$ space.*

4 Valid query answers

In our paper we use the negation-free fragment of XPath 1.0 [26] restricted to its logical *core* (only element and text nodes, and only functions selecting the string value of nodes). Our approach, however, is applicable to a wider class of negation-free Regular XPath Queries [18]. We assume the standard semantics of XPath queries and by $QA^Q(T)$ we denote the answers to the query Q in the tree T.

We use an evaluation method that is geared towards the computation of valid answers. The basic notion is this of a tree fact (n, p, x) which states that an *object* x (a node, a label, or a string constant) is reachable from the node n with an XPath expression p.

We distinguish *basic* tree facts which use only $/*$, $following\text{-}sibling{::}*$, $name(.)$, and $text(.)$ path expressions. We note that basic tree facts are capable to represent XML trees. For instance let n_0, n_1, n_2 be the root, the first and the second child of the root in the document $T_2 = \mathtt{A}(\mathtt{B}(), \mathtt{c})$, then examples of basic facts are: $(n_1, name(.), \mathtt{B})$, $(n_0, /*, n_1)$, $(n_1, following\text{-}sibling{::}*, n_2)$, and $(n_2, text(.), \mathtt{c})$. The facts that involve complex XPath expressions can be derived from the basic facts. For example from the basic facts for T_2 we can derive $(n_0, /B/following\text{-}sibling{::}text(.), \mathtt{c})$. We note that the derivation process for negation-free XPath expressions is *monotonic*, i.e. introducing new (basic) facts cannot invalidate facts previously derived.

Given a document T and a query Q we construct the set of all relevant tree facts B by adding to B all the basic facts of T. If adding a fact allows to derive new facts involving any subexpression of Q, these facts are also added to B (B is closed under subexpressions of Q). To find the answers to Q we simply select the facts that originate in the root of T and involve Q.

4.1 Validity-sensitive query evaluation

Definition 4 (Valid query answers). *Given a tree T, a query Q, and a DTD D, an object x is a* valid answer *to Q in T w.r.t D if x is an answer to Q in every repair of T w.r.t. D. We denote the set of all valid query answers to Q in T w.r.t. D by $VQA_D^Q(T)$.*

Computing valid query answers We construct a bottom-up algorithm that for every node constructs the set of *certain* tree facts that hold in every repair of the subtree rooted in this node. The set of certain tree facts computed for the root node is used to obtain the valid answers to the query (similarly to standard answers).

We now fix the tree T, the DTD D, and the query Q, and assume that we have constructed the trace graph U_T^* for T as described in Section 3. We also assume that the sets of certain tree facts for the children of the root of T have been computed earlier.

Recall that the macro operation Ins_Y corresponds to inserting a minimum-size tree valid w.r.t. the DTD, whose root label is Y. Thus for every label Y our algorithm needs the set C_Y of (certain) tree facts present in every minimal valid tree with the root's label Y. We note that this set can be constructed with a simple algorithm (omitted here).

4.2 Naive computation of valid answers

We start with a naive solution, which may need an exponential time for computation. Later on we present a modification which guarantees a polynomial execution time.

For each repairing path in U_T^* the algorithm constructs the set of certain tree facts present in every repair corresponding to this path. Assume now that $T = X(T_1, \ldots, T_n)$, and the root nodes of $T, T_1, \ldots, T_n$ are respectively $r, r_1, \ldots, r_n$.

For a path $q_0^0 = v_0, v_1, \ldots, v_m$ in U_T^* we compute the corresponding set C of certain facts in an incremental fashion (in every step we keep adding any facts that can be derived for subexpressions of the query Q):

1. for q_0^0 the set of certain facts consists of all the basic fact for the root node;
2. if C is the set corresponding to $v_0, \ldots, v_{k-1}$, then the set C' corresponding to $v_0, \ldots, v_k$ is obtained by one of the 3 following cases depending on the type of edge from v_{k-1} to v_k:
   - for $q^{i-1} \xrightarrow{Del} q^i$ no additional facts are added, i.e., $C' = C$;
   - for $q^{i-1} \xrightarrow{Rep} p^i$ we *append* the tree facts of T_i to C, i.e., we add to C certain facts of the tree T_i with the basic fact $(r, /*, r_i)$; if on the path $v_0, \ldots, v_{k-1}$ other trees have been appended (with either Rep or Ins_Y instruction), then we also add the fact $(r', following\text{-}sibling :: *, r_i)$ where r' is the root node of the last appended tree;
   - $q^i \xrightarrow{Ins_Y} p^i$ is treated similarly to the previous case, but we append (a copy of) C_Y.

Naturally, the set of certain facts for T is the intersection of all sets corresponding to repairing paths in U_T^*.

If we implement this algorithm so that in every vertex v of U_T^* we store the collection of all sets of certain tree facts that corresponds to every path from q_0^0 to v, then we can use the following *eager intersection* optimization:

> Let C_1 and C_2 be two sets of certain tree facts corresponding to different paths from q_0^0 to a vertex v. Suppose that $v \longrightarrow v'$ and the edge is labeled with an operation that appends some tree (either Rep or Ins_Y). Let C_1' and C_2' be the sets for v' obtained from respectively C_1 and C_2 as described before. Instead of storing in v' both sets C_1' and C_2' we only store their intersection $C_1' \cap C_2'$.

With a simple induction over the column number we show that the number of sets of tree facts stored in a vertex in the i-th column is $O(i \times |Q| \times |\Sigma|)$.

We use the notion of *data complexity* [24] which allows to express the complexity of the algorithm in terms of the size of the document only (by assuming other input components to be fixed).

Theorem 3. *The data-complexity of computation of valid answers to negation-free core XPath queries is PTIME.*

We note that including query into the input leads to intractability The proof, skipped here, uses the repairs from Example 5 to represent valuations of n boolean variables and reduces the complement of SAT to computing the valid query answer to a core XPath query obtained from a simple translation of the CNF formula. This result shows that computing valid query answers is considerably more complex than computation of standard answers, whose combined complexity is known to be polynomial [13].

5 Experimental evaluation

In our experiments, we tested 2 algorithms: DIST computing $dist(D,T)$ and VQA computing valid answers. We compared these algorithms with an algorithm VALIDATE for validation of a document and an algorithm QA computing standard answers. All compared algorithms have been implemented using a common set of programming tools including: the parser, the representation of regular expressions and corresponding NDFA's, the representation for tree facts, and algorithms maintaining closure of the set of tree facts. For ease of implementation, we considered only a restricted class of *non-ascending path queries* which use only simple filter conditions (testing tag and text elements), do not use union, and involve only $child$, $descendant$, and $following\text{-}sibling$ axises. We note that those queries are most commonly used in practice and the restrictions allow to construct algorithms that compute standard answers to such queries in time linear in the size of the document. This is also the complexity of the QA algorithm.

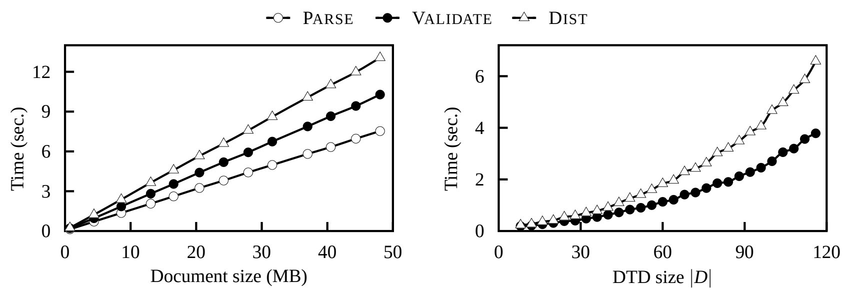
Data generation To observe the impact of the document size on the performance of algorithms, we randomly generated a valid document and we introduced the violations of validity to a document by randomly removing and inserting nodes. To measure the invalidity of a document T we use the *invalidity ratio* $dist(T,D)/|T|$. All the documents used for tests had a small height, 8-10.

For most of the experiments we used the DTD D_0 and the query Q_0 from Example 1. To measure the impact of the DTD size on the performance, we generated a family of DTDs D_n, $n \geq 0$: $D_n(\mathtt{A}) = (\ldots((\mathtt{PCDATA} + \mathtt{A}_1) \cdot \mathtt{A}_2 + \mathtt{A}_3) \cdot \mathtt{A}_4 + \ldots \mathtt{A}_n)^*$,$D_n(\mathtt{A}_i) = \mathtt{A}^*$. For those documents we used a simple query $// * /text(.)$.

Environment The system was implemented in Java 5.0 and tested on an Intel Pentium M 1000MHz machine running Windows XP with 512 MB RAM and 40 GB hard drive.

5.1 Experimental results

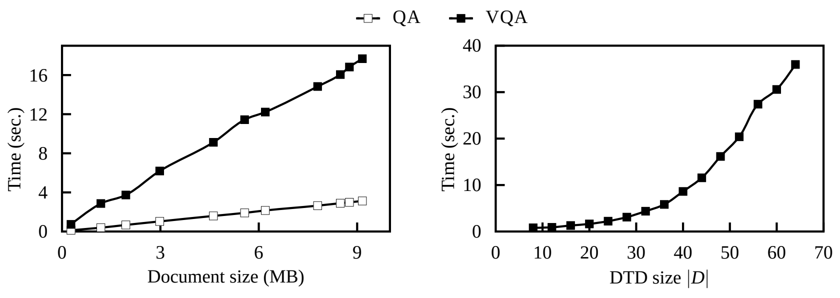
Results in Figure 4(a) and in Figure 4(b) confirm our analysis: edit distance between a document and a DTD can be computed in time linear in the document size and quadratic in the size of the DTD. If we take as the base line the time needed to parse the whole file (PARSE), then we observe that the overhead needed to perform computations is small. Because our approach to computing edit distance doesn't assume any particular properties of the automata used to construct the trace graph, Figure 4(b) allows us to make the following conjecture: Any techniques that optimize the automata to efficiently validate XML documents should also be applicable to the algorithm for computing the distance of XML documents to DTDs.



(a) for variable document size (b) for variable DTD size

Fig. 4. Edit distance computation time (0.1% invalidity ratio)

Figure 5(a) shows that for the DTD D_0 computing valid query answers is about 6 times longer than computing query answers with QA. Similarly to computing edit distance, computing valid answers involves constructing the restoration graph. This explains the quadratic dependency between the performance time and the size of DTD observed for VQA in Figure 5(b).



(a) for variable document size (b) for variable DTD size

Fig. 5. Valid answers computation time (0.1% invalidity ratio).

6 Related work

Tree edit distance Tree edit distance is a generalization of the classical string edit distance. There are several different versions of the former notion varying with the semantics of tree operations [7]. In the most studied approach [23], the deleting operation also can be performed on a internal node, in which case the children are *promoted up*. Conversely, the inserting operation can *push down* a contiguous sequence of nodes. The implied notion of edit distance is not equivalent to ours (our notion is sometimes called *1-degree edit distance* [22]). In the area of data integration, insertions and deletions of internal document nodes could be used for the resolution of major structural discrepancies between documents. However, such operations require shifting nodes between levels and thus it is not clear if our approach can be adapted to that context. The notion of edit distance identical to ours has been used in papers dealing with the maintenance of XML integrity [1, 5, 6] and to measure structural similarity between XML documents [20]. [9] studies an extension of the basic tree edit framework with *moves*: a subtree can be shifted to a new location. In the context of validity-sensitive querying, extending our approach with move operations would allow to properly handle situations where invalidity of the document is caused by transposition of elements.

Almost every formulation of edit distance, including ours, allows to assign a non-unit cost to each operation.

Structural restoration A problem of correcting a *slightly invalid* document is considered in [9]. Under certain conditions, the proposed algorithm returns a valid document whose distance from the original one is guaranteed to be within a multiplicative constant of the minimum distance. The setting is different from ours: XML documents are encoded as binary trees, so performing editing operations on a encoded XML document may shift nodes between levels in terms of the original tree.

A notion equivalent to the distance of a document to a DTD (Definition 2) was used to construct error-correcting parsers for context-free languages [2].

Consistent query answers for XML [10] investigates querying XML documents that are valid but violate functional dependencies. Two repairing actions are considered: updating element values with a *null* value and marking nodes as unreliable. This choice of actions prevents from introducing invalidity in the document upon repairing it. Nodes with null values or marked as unreliable do not cause violations of functional dependencies but also are not returned in the answers to queries. Repairs are consistent instances with a minimal set of nodes affected by the repairing actions.

A set of operations similar to ours is considered for consistent querying of XML documents that violate functional dependencies in [11]. Depending on the operations used different notions of repairs are considered: *cleaning* repairs obtained only by deleting elements, *completing* repairs obtained by inserting nodes, and *general* repairs obtained by both operations.

[25] is another adaptation of consistent query answers to XML databases closely based on the framework of [4].

7 Conclusions and Future work

In this paper we investigated the problem of querying XML documents containing violations of validity of a local nature caused by missing or superfluous nodes. We proposed a framework which considers possible repairs of a given document obtained by applying a minimum number of operations that insert or delete nodes. We demonstrated algorithms for (a) measuring invalidity in terms of document-to-DTD distance, and (b) validity-sensitive querying based on the notion of valid query answer.

We envision several possible directions for future work. First, one can investigate if valid answers can be obtained using query rewriting [14]. Second, it is an open question if negation could be introduced into our framework. Third, it would be of significant interest to establish a complete picture of how the complexity of the computational problems considered in this paper (computing document-to-DTD distance, computing valid query answers) depends on the query language and the repertoire of the available tree operations (other operations include subtree swap, restricted subtree move). Finally, it would be interesting to find out to what extent our framework can be adapted to handle semantic inconsistencies in XML documents, for example violations of key dependencies.

References

1. S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *International Conference on Very Large Data Bases (VLDB)*, pages 38–49, 1998.
2. A. V. Aho and T. G. Peterson. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM Journal on Computing*, 1(4):305–312, 1972.
3. S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *International Conference on Extending Database Technology (EDBT)*, pages 496–513, 2002.
4. M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.

5. A Balmin, Y. Papakonstantinou, and V. Vianu. Incremental Validation of XML Documents. *ACM Transactions on Database Systems*, 29(4):710–751, December 2004.
6. M. Benedikt, W. Fan, and F. Geerts. XPath Satisfiability in the Presence of DTDs. In *ACM Symposium on Principles of Database Systems (PODS)*, 2005.
7. P. Bille. Tree Edit Distance, Aligment and Inclusion. Technical Report TR-2003-23, The IT University of Copenhagen, 2003.
8. P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *ACM SIGMOD*, 2005.
9. U. Boobna and M. de Rougemont. Correctors for XML Data. In *International XML Database Symposium (Xsym)*, pages 97–111, 2004.
10. S. Flesca, F. Furfaro, S. Greco, and E. Zumpano. Repairs and Consistent Answers for XML Data with Functional Dependencies. In *International XML Database Symposium (Xsym)*, volume 2824 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2003.
11. S. Flesca, F Furfaro, S. Greco, and E. Zumpano. Querying and Repairing Inconsistent XML Data. In *Web Information Systems Engineering (WISE)*, pages 175–188, 2005.
12. A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient Management of Inconsistent Databases. In *ACM SIGMOD International Conference on Management of Data*, 2005.
13. G. Gottlob, C. Koch, and R. Pichler. XPath Processing in a Nutshell. *SIGMOD Record*, 2003.
14. G. Grahne and A. Thomo. Query Answering and Containment for Regular Path Queries under Distortions. In *Foundations of Information and Knowledge Systems (FOIKS)*, pages 98–115, 2004.
15. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2001.
16. A. Klarlund, T. Schwentick, and D. Suciu. XML: Model, Schemas, Types, Logics and Queries. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*. Springer-Verlag, 2003.
17. W. L. Low, W. H. Tok, M. Lee, and T. W. Ling. Data Cleaning and XML: The DBLP Experience. In *International Conference on Data Engineering (ICDE)*, page 269, 2002.
18. M. Marx. Conditional XPath, the First Order Complete XPath Dialect. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 13–22, 2004.
19. F. Neven. Automata, Logic, and XML. In *Workshop on Computer Science Logic (CSL)*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.
20. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Workshop on the Web and Databases (WebDB)*, pages 61–66, 2002.
21. P. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Approximate XML Query Answers. In *ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2004.
22. S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6):184–186, 1977.
23. D. Shasha and K. Zhang. Approximate Tree Pattern Matching. In A. Apostolico and Z. Galil, editors, *Pattern Matching in Strings, Trees, and Arrays*, pages 341–371. Oxford University Press, 1997.
24. M. Y. Vardi. The Complexity of Relational Query Languages. In *ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.
25. S. Vllalobos. Consistent Answers to Queries Posed to Inconsistent XML Databases. Master's thesis, Catholic University of Chile (PUC), 2003. In Spanish.
26. W3C. XML path language (XPath 1.0), 1999.

XQuery!: An XML query language with side effects

Giorgio Ghelli[1], Christopher Ré[2], and Jérôme Siméon[3]

[1] Università di Pisa
[2] University of Washington
[3] IBM T.J. Watson Research Center

Abstract. As XML applications become more complex, there is a growing interest in extending XQuery with side-effect operations, notably XML updates. However, the presence of side-effects is at odds with XQuery's declarative semantics in which evaluation order is unspecified. In this paper, we define "XQuery!", an extension of XQuery 1.0 that supports first-class XML updates and user-level control over update application, preserving the benefits of XQuery's declarative semantics when possible. Our extensions can be easily implemented within an existing XQuery processor and we show how to recover basic database optimizations for such a language.

1 Introduction

As XML applications grow in complexity, developers are calling for advanced features in XML query languages. Many of the most requested extensions, such as XML updates, support for references, and variable assignment, involve side-effects. So far, proposed update extensions for XQuery [14, 19, 21, 1] have been based on restricted compositionality and a "snapshot semantics", where updates are only applied at the end of query execution. This approach preserves as much of XQuery's declarative semantics as possible, but the query cannot use the result of an update for further processing, limiting expressiveness in a way which is not always acceptable for applications.

In this paper, we develop the semantic foundations for extending XQuery 1.0 with side-effect operations in a fully compositional way. We use that framework to define XQuery! (read: "XQuery Bang"), an extension of XQuery 1.0 [2] that supports compositional XML updates and user-level control over update application. We show such a language can be obtained with limited impact on XQuery's declarative semantics and classical optimization techniques. To the best of our knowledge, this is the first complete treatment and implementation of a compositional side-effect extension of XQuery. The semantic framework is characterized by the presence of an operator (`snap`) that allows users to identify declarative fragments within their side-effecting programs, and which enables the recovery of traditional database optimizations.

XQuery! supports the same basic update operations as previous proposals [14, 7, 6]. However, the ability to use updates in any context (e.g., in function calls) and to control update application makes it more expressive than previous proposals. Compositionality is one of the main design principles in XQuery 1.0, resulting in a language simpler to explain to users and specify. Our experience with a more restricted update language [21] shows that applications often require the additional expressiveness. We illustrate how compositionality between queries and updates in XQuery! can be used to develop a simple Web service that includes logging of service calls.

The contributions of this paper are:

– A formal description of a semantic framework for extending XML query languages with side-effect operations which can appear anywhere in the query.
– The description of a new construct (`snap`) that can be used to control update application. The semantics of `snap` enables unlimited nesting and allows the optimizer to recover standard database optimizations, even in the presence of side-effects.
– The definition of XQuery!, an extension to XQuery 1.0 with first-class updates, and an example of its use in a Web service usecase.
– The description of a complete implementation of XQuery!. We show that such an implementation can easily be obtained from an existing XQuery engine.

The main novelty in our framework lies in the ability to control update application through the `snap` construct. The notion of delaying update application to the end of query evaluation (so called *snapshot* semantics) was first proposed in [19, 14], and has been studied further in [7, 6, 1]. Previous proposals apply that approach to the whole query, while XQuery! provides programmer control of the snapshot scope through the `snap` operator. Languages with explicit control of the snapshot semantics are mentioned explicitly in the XQuery update requirements document [4], and have been explored by the W3C XML update task force [9, 3]. Work on the XL programming language [10] indicates support for fully compositional updates, but not for control of update application. To the best of our knowledge, our work is the first to propose a complete treatment of such an operator, and to explicit its relationship with optimization properties of the language.

Due to space limitations, we restrict the presentation to the main aspects of the language and its semantics. We first introduce XQuery! through a Web service usecase, before giving a formal definition of the language semantics. We then give an overview of our implementation, and discuss optimization issues. More details of the language, its complete formal semantics and more details about the implementation can be found in the complete paper [11].

2 XQuery! Use Case: Adding Logging to an XQuery Web Service

2.1 Snapshot semantics

Before we illustrate the use of XQuery!, we introduce the notion of snapshot semantics. All the update extensions to XQuery we are aware of [19, 14, 7, 6, 1] delay update applications up to the end of query execution, in order to retain the declarative semantics of XQuery. For instance, consider the following query which inserts a new `buyer` element for each person who buys an item.

```
for $p in $auction//person
for $t in $auction//closed_auction
where $t/buyer/@person = $p/@id
return insert { <buyer name="{$p/name}"
                       itemid="{$t/itemref/@item}" /> }
       into { $purchasers }
```

This is a typical join query, and the snapshot semantics ensures that traditional optimization techniques, such as algebraic rewritings and lazy evaluation, can be applied.

In XQuery!, where the snapshot semantics is controlled explicitly, the absence of any internal `snap` allows similar optimizations. We come back to this point in more details in Section 4.

In addition, in order to facilitate rewritings, previous proposals limit the use of updates to specific sub-expressions, typically in the return clause of a FLWOR, as in the above example. In the rest of this section, we give a more advanced Web service use-case which requires complex composition of side-effects and queries, and control over update application.

2.2 The Web Service scenario: updates inside functions

We assume a simple Web service application in which service calls are implemented as XQuery functions organized in a module. Because of space limitations, we focus on the single function `get_item`, which, given an itemid and the userid of the requester, returns the item with the given itemid; the userid is ignored for now. The server stores the auction document from XMark [20] in a variable `$auction`. The following is a possible implementation for that function using standard XQuery.

```
declare function get_item($itemid,$userid) {
  let $item := $auction//item[@id = $itemid]
  return $item
};
```

Now, let's assume that the Web service wants to log each item access. This can be easily done in XQuery! by adding an insert operation in the body of the function.

```
declare function get_item($itemid,$userid) {
  let $item := $auction//item[@id = $itemid]
  return (
    ( let $name := $auction//person[@id = $userid]/name return
      insert { <logentry user="{$name}" itemid="{$itemid}"/> }
      into { $log }),
    $item
  )
};
```

This simple example illustrates the need for expressions that have a side-effect (the log entry insertion) and also return a value (the item itself).

Note that in the above example we use XQuery's sequence construction (`,`) to compose the conditional insert operation with the result `$item`. This is a convenience made possible by the fact that the value returned by atomic update operations is always the empty sequence.

2.3 Controlling update application

The other central feature of our approach is the ability to control the "snapshot scope". A single global scope is often too restrictive, since many applications, at some stage of their computation, need to see the result of previous side-effects. For this reason, XQuery! supports a `snap {` *Expr* `}` operator which evaluates *Expr*, collects its update requests, and makes the effects of these updates visible to the rest of the query. A `snap` is always implicitly present around the top-level query in the main XQuery! module, so that the usual "delay until the end" semantics is obtained by default. However, when needed, the code can decide to see its own effects. For example, consider the following simple variant for the logging code, where the log is summarized into an archive once every *$maxlog* insertions. (`snap insert{} into{}` abbreviates `snap{insert{} into{}}`, and similarly for the other update primitives).

```
( let $name := $auction//person[@id = $userid]/name
  return
   (snap insert { <logentry user="{$name}"
                           itemid="{$item/@id}"/> }
         into { $log },
    if (count($log/logentry) >= $maxlog)
    then (archivelog($log,$archive),
          snap delete $log/logentry )
    else ())),
```

Here, the `snap` around `insert` makes the insertion happen. The insertion is visible to the code inside the subsequent if-then-else because XQuery! semantics imposes that in the sequence constructor `(e1,e2)`, `e1` be evaluated before `e2`. Hence, XQuery!'s ability to support the above example relies on the combination of the `snap` operator and of an explicit evaluation order. This is an important departure from XQuery 1.0 semantics, and is discussed in the next subsection.

In many situations, different scopes for the `snap` would lead to the same result. In such cases, the programmer can adopt a simple criterion: make `snap` scope as broad as possible, since a broader `snap` favors optimization. A `snap` should only be closed when the rest of the program relies on the effect of the updates.

2.4 Sequence order, evaluation order, and update order

In XQuery 1.0, queries return sequences of items. Although sequences of items are ordered, the evaluation order for most operators is left to the implementation. For instance, in the expression (e_1, e_2), if e_1 and e_2 evaluate respectively to v_1 and v_2, then the value of e_1, e_2 must be v_1, v_2, in this order. However, the engine can evaluate e_2 before e_1, provided the result is presented in the correct sequence order. This freedom is visible for instance, if both expressions e_1 and e_2 were to raise an error, as which of those errors is reported may vary from implementation to implementation.

Although that approach is reasonable in an almost-purely functional language as XQuery 1.0, it is widely believed that programs with side-effects are impossible to reason about unless the evaluation order is easy to grasp.[4] For this reason, in XQuery! we adopt the standard semantics used in popular functional languages with side-effects [16, 15], based on the definition of a precise evaluation order. This semantics is easy to understand for a programmer and easy to formalize using the XQuery 1.0 formal semantic

[4] Simon Peyton-Jones: "lazy evaluation and side effects are, from a practical point of view, incompatible" [13].

style, but is quite constraining for the compiler. However, as we discuss in Section 3, inside an innermost `snap` no side-effect takes place, hence we recover XQuery 1.0 freedom of evaluation order in those cases. In other words, inside an innermost `snap`, both the pure subexpressions and the update operations can be evaluated in any order, provided that, at the end of the `snap` scope, both the item sequence and the list of update requests are presented in the correct order.

The order of update requests is a bit harder to maintain than sequence order, since a FLWOR expression may generate updates in the *for*, *where,* and *return* clause, while result items are only generated in the *return* clause. For this reason, XQuery! supports alternative semantics for update application, discussed in Section 3.2, which do not depend on order.

2.5 Nested snap

Support for nested snap is central to our proposal, and is essential for compositionality. Assume, for example, that a counter is implemented using the following function.

```
declare variable $d := element counter { 0 };
declare function nextid() as xs:integer {
  snap { replace { $d/text() } with { $d + 1 },
         $d }
};
```

The snap around the function body ensures that the counter function performs as expected, returning an increasing value after each call. Obviously, the `nextid()` function may be used in the scope of another snap. For instance, the following variant of the logging code computes a new id for every log entry.

```
  (::: Logging code :::)
  ( let $name := $auction//person[@id = $userid]/name
    return
     (snap insert { <logentry id="{nextid()}"
                              user="{$name}"
                              itemid="{$item/@id}"/> }
           into { $log },
      if (count($log/logentry) >= $maxlog) ...
  (::: End logging code :::)
```

The example shows that the `snap` operator must not freeze the state when its scope is opened, but just delay the updates that are in its immediate scope until it is closed. Any nested snap opens a nested scope, and makes its updates visible as soon as it is closed. The details of this semantics are explained in Section 3.

3 XQuery! Semantics

The original semantics of XQuery is defined in [5] as follows. First, each expression is normalized to a *core* expression. Then, the meaning of core expressions is defined by a semantic judgment $\mathit{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{value}$. This judgment states that, in the dynamic context *dynEnv*, the expression *Expr* yields the value *value*, where *value* is an instance of the XQuery data model (XDM).

To support side-effect operations, we extend the data model with a notion of store that maintains the state of the instances that are being processed. It should be clear from the discussion in Section 2 that only snap expressions actually modify the store. We extend the semantic judgment so that expressions may modify the store, and produce both a value and a list of pending updates. In the rest of this section, we introduce the update primitives supported by the XQuery! language, followed by the data model extensions. We then shortly describe normalization, and finally define the new semantic judgment.

3.1 Update primitives

At the language level, XQuery! supports a set of standard updates primitives: insertion, deletion, replacement, and renaming of XML nodes [14, 19, 21, 1, 7, 6]. The language also includes an explicit deep-copy operator, written `copy { ... }`. The full grammar for the XQuery! extension to XQuery 1.0 is given in [11].

The detailed semantics of these primitives is also standard: insertion allows a sequence of nodes to be inserted below a parent at a specified position. Replacement allows a node to be replaced by another, and renaming allows the node name to be updated. Finally, to better deal with aliasing issues in the context of a compositional language, the semantics of the delete operation does not actually delete nodes, but merely *detaches* nodes from their parents. If a “deleted” (actually, detached) node is still accessible from a variable, it can still be queried, or inserted somewhere. Although this approach requires garbage collection, we believe it is slightly simpler to specify, implement, and program with than the alternative “erase” semantics.

3.2 XDM stores and update requests

Store. To represent the state of XQuery! computation, we need a notion of *store*, which specifies the valid node ids and, for each node id, its kind (element, attribute, text...), parent, name, and content. A formal definition can be found in [11, 12, 8]. On this store, we define accessors and constructors corresponding to those of the XDM. Note that this presentation focuses on well-formed documents, and does not consider the impact of types on the data model representation and language semantics.

Update requests. We then define, for each XQuery! update primitive, the corresponding *update request*, which is a tuple that contains the operation name and its parameters, written as “opname(par_1,...,par_n)”. For each update request, its *application* is a partial function from stores to stores. The application of “insert (*nodeseq*,*nodepar*,*nodepos*)” inserts all nodes of *nodeseq* as children of *nodepar*, after *nodepos*. For each update request we also define some preconditions for its parameters. In the insert case, they include the fact that nodes in *nodeseq* must have no parent, and that *nodepos* must be a child of *nodepar*. When the preconditions are not met, the update application is undefined.

Update lists. An update list, denoted Δ, is a list of update requests. Update lists are collected during the execution of the code inside a `snap`, and are applied when the `snap` scope is closed. An update list is an *ordered* list, whose order is fully specified by the language semantics.

Applying an update list to the store. For optimization reasons, XQuery! supports three semantics for update list application: *ordered*, *non-deterministic*, and *conflict-detection*. The programmer chooses the semantics through an optional keyword after each `snap`.

In the *ordered* semantics, the update requests are applied in the order specified by Δ. In the *non-deterministic* semantics, the update requests are applied in an arbitrary order. In the *conflict-detection* semantics, update application is divided into conflict verification followed by store modification. The first phase tries to prove, by some simple rules, that the update sequence is actually conflict-free, meaning that the ordered application of every permutation of Δ would produce the same result. If verification fails, update application fails. If verification succeeds, the store is modified, and the order of application is immaterial.

The *ordered* approach is simple and deterministic, but imposes more restrictions on the optimizer. The *non-deterministic* approach is simpler to implement and optimize, but makes code development harder, especially in the testing phase. Finally, the *conflict-detection* approach gives the optimizer the same re-ordering freedom as the non-deterministic approach while avoiding non-determinism. However, it rules out many reasonable pieces of code, as exemplified in the full paper. Moreover, it can raise run-time failures which may be difficult to understand and to prevent.

Our implementation currently supports all the three semantics. We believe more experience with concrete applications is needed in order to assess the best choice.

3.3 Normalization

Normalization simplifies the semantics specification by first transforming each XQuery! expression into a *core* expression, so that the semantics only needs to be defined on the core language. The syntax of XQuery! core for update operations is almost identical to that of the surface language. The only non-trivial normalization effect is the insertion of a deep copy operator around the first argument of `insert`, as specified by the following normalization rule; the same happens to the second argument of `replace`. As with element construction in XQuery 1.0, this copy prevents the inserted tree from having two parents.

$$\frac{[\![\texttt{insert }\{\mathit{Expr}_1\}\texttt{ into }\{\mathit{Expr}_2\}]\!]}{\texttt{insert }\{\texttt{copy }\{[\![\mathit{Expr}_1]\!]\}\}\texttt{ as last into }\{[\![\mathit{Expr}_2]\!]\}}$$

3.4 Formal semantics

Dynamic evaluation judgment. We extend the semantic judgment "$\mathit{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{value}$", in order to deal with delayed updates and side-effects, as follows:

$$\mathit{store}_0; \mathit{dynEnv} \vdash \mathit{Expr} \Rightarrow \mathit{value}; \Delta; \mathit{store}_1$$

Here, store_0 is the initial store, *dynEnv* is the dynamic context, *Expr* is the expression being evaluated, *value* and Δ are the value and the list of update requests returned by the expression, and store_1 is the new store after the expression has been evaluated. The updates in Δ have not been applied to store_1 yet, but *Expr* may have modified store_1 thanks to a nested `snap`, or by allocating new elements.

Observe that, while the store is modified, the update list Δ is just returned by the expression, exactly as the *value*. This property hints at the fact that an expression which just produces update requests, without applying them, is actually side-effects free, hence can be evaluated with the same approaches used to evaluate pure functional expressions. This is the main reason to use a snapshot semantics: inside the innermost `snap`, where updates are collected but not applied, lazy evaluation techniques can be applied.

Dynamic semantics of XQuery expressions. The presence of stores and Δ means that every judgment in XQuery 1.0 must be extended in order to properly deal with them. Specifically, every semantic judgment which contains at least two subexpressions has to be extended in order to specify which subexpression has to be evaluated first. Consider for example the XQuery! rule for the sequence constructor.

$$\frac{\begin{array}{c} store_0;\, dynEnv \vdash Expr_1 \Rightarrow value_1;\, \Delta_1;\, store_1 \\ store_1;\, dynEnv \vdash Expr_2 \Rightarrow value_2;\, \Delta_2;\, store_2 \end{array}}{store_0;\, dynEnv \vdash Expr_1, Expr_2 \Rightarrow value_1, value_2;\, (\Delta_1, \Delta_2);\, store_2}$$

As written, $Expr_1$ must be evaluated first in order for $store_1$ to be computed and passed for the evaluation of $Expr_2$.

If a sub-expression is guaranteed not to invoke a `snap`, the compiler can again choose evaluation order as in the original XQuery 1.0 semantics for that sub-expression. Of course, Δ_1 must precede Δ_2 in the result, when the *ordered* approach is followed, but this is not harder than preserving the order of ($value_1$, $value_2$); preserving update order is more complex in the case of FLWOR expressions and function calls (see [11]).

Dynamic semantics of XQuery! operations. We have to define the semantics of `copy`, of the update operators, and of `snap`. `copy` just invokes the corresponding operation at the data model level, adding the corresponding nodes to the store. The evaluation of an update operation produces an update request, which is added to the list of the pending update requests produced by the subexpressions, while `replace` produces *two* update requests, insertion and deletion. Here is the semantics of `replace`. The metavariables express constraints on rule applicability: *node* and *nodepar* can only be matched to node ids, and *nodeseq* only to a sequence of node ids.

$$\frac{\begin{array}{c} store_0;\, dynEnv \vdash Expr_1 \Rightarrow node;\, \Delta_1;\, store_1 \\ store_1;\, dynEnv \vdash Expr_2 \Rightarrow nodeseq;\, \Delta_2;\, store_2 \\ store_2;\, dynEnv \vdash \mathrm{parent}(node) \Rightarrow nodepar;\, ();\, store_2 \\ \Delta_3 = (\Delta_1, \Delta_2, \mathrm{insert}(nodeseq, nodepar, node), \mathrm{delete}(node)) \end{array}}{store_0;\, dynEnv \vdash \texttt{replace \{}Expr_1\texttt{\} with \{}Expr_2\texttt{\}} \Rightarrow ();\, \Delta_3;\, store_2}$$

The evaluation produces an empty sequence and an update list Δ_3. It may also modify the store, but only if either $Expr_1$ or $Expr_2$ modify it. If they only perform allocations or copies, their evaluation can still be commuted or interleaved. If either executes a `snap`, the processor must follow the order specified by the rule, since, for example, $Expr_2$ may depend on the part of the store which has been modified by a `snap` in $Expr_1$. The two update requests produced by the operation are just inserted into the pending update list Δ_3 after every update requested by the two subexpressions. The actual order is only relevant if the *ordered* semantics has been requested for the smallest enclosing `snap`.

The rule for `snap` $\{Expr\}$ looks very simple: *Expr* is evaluated, it produces its own update list Δ, Δ is applied to the store, and the value of *Expr* is returned.

$$\frac{\begin{array}{c} store_0; dynEnv \vdash Expr \Rightarrow value; \Delta; store_1 \\ store_2 = \text{apply } \Delta \text{ to } store_1 \end{array}}{store_0; dynEnv \vdash \texttt{snap}\ \{Expr\} \Rightarrow value; (); store_2}$$

The evaluation of *Expr* may itself modify the store, and `snap` updates this modified store. For example, the following piece of code inserts `<b/><a/><c/>` into `$x`, in this order, since the internal `snap` is closed first, and it only applies the updates in its own scope.

```
snap ordered { insert {<a/>} into $x,
               snap   { insert {<b/>} into $x },
               insert {<c/>} into $x }
```

Hence, the formal semantics implicitly specifies a stack-like behavior, reflected by the actual stack-based implementation that we adopted (see [11]).

In the appendix we list the semantic rules for the other update operations, and for the most important core XQuery 1.0 expressions.

4 Implementation and Optimization

XQuery! has been implemented as an extension to the Galax XQuery engine [18, 17], and a preliminary version is available for download[5]. In this section, we review the modifications that were required to the original Galax compiler to support side-effects, notably changes to the optimizer.

4.1 Data model and run-time

Changes to the data model implementation to support atomic updates were not terribly invasive. The only two significant challenges relate to dealing with document order maintenance, and garbage collection of persistent but unreachable nodes, resulting from the detach semantics. Both of these aspects are beyond the scope of this paper.

The run-time must be modified to support update lists, which are computed in addition to the value for each expression. The way the update lists are represented internally depends on whether the `snap` operator uses the ordered semantics or not (See Section 3.2). Because the nondeterministic and conflict-detection semantics are both independent of the actual order of the atomic updates collected in a `snap` scope, they can be easily implemented using a stack of update lists, where each update list on the stack corresponds to a given `snap` scope, and where the order inside a list is irrelevant. The invocation of an update operation adds an update to the update list on the top of the stack. When exiting a `snap`, the top-most update list is popped from the stack and

[5] `http://xquerybang.cs.washington.edu/`

applied. In the case of conflict-detection semantics, it is also checked for conflicts, in linear time, using a pair of hash-tables over node ids.

This implementation strategy has the virtue that it does not require substantial modifications to an existing compiler. The implementation of the ordered semantics is more involved, as we must rely on a specialized tree structure to represent the update lists in a way that allows the compiler to lazily evaluate FLWOR expressions and still retain the order in which each update must be applied. We refer to the full paper [11] for more details.

4.2 Compilation architecture

Implementing XQuery! does not require modifications to the XQuery processing model. The XQuery! compiler in Galax proceeds by first *parsing* the query into an AST and *normalization*, followed by a phase of syntactic *rewriting*, *compilation* into the XML algebra of [18] with some simple update extensions, *optimization* and finally *evaluation*.

Changes to parsing and normalization are trivial (See Section 3). To preserve the XQuery! semantics, some of the syntactic rewritings must be guarded by a judgment checking whether side effects may occur in a given sub-expression. Of course, this is not necessary when the query is guarded by an innermost `snap`, i.e., a `snap` whose scope contains no other `snap`, nor any call to any function which may cause a `snap` to be evaluated. Inside such innermost `snap`, all the rewritings immediately apply.

4.3 XQuery! optimizer

Galax uses a rule-based approach in several phases of the logical optimization. Most rewrite rules require some modifications. To illustrate the way the optimizer works, let us consider the following variant of XMark query 8 which, for each person, stores information about the purchased items.

```
for $p in $auction//person
let $a :=
  for $t in $auction//closed_auction
  where $t/buyer/@person = $p/@id
  return (insert { <buyer person="{$t/buyer/@person}"
                          itemid="{$t/itemref/@item}" /> }
          into { $purchasers }, $t)
return <item person="{ $p/name }">{ count($a) }</item>
```

Ignoring the insert operation for a moment, the query is essentially the same as XMark 8, and can be evaluated efficiently with an outer join followed by a group by. Such a query plan can be produced using query unnesting techniques such as those proposed in e.g., [18]. A naive nested-loop evaluation has complexity $O(|person| * |closed\_auction|)$. Using an outer join/group by with a typed hash join, we can recover the join complexity of $O(|person| + |closed\_auction| + |matches|)$, resulting in a substantial improvement.

In XQuery!, recall that the query is always wrapped into a top-level snap. Because that top-level snap does not contain any nested snap, the state of the database will not change during the evaluation of the query, and a outer-join/group-by plan can be used. The optimized plan generated by our XQuery! compiler is shown below. The syntax used for that query plan is that of [18], where algebraic operators are written as follows:

$\mathsf{Op}[\mathsf{p}_1,\ldots,\mathsf{p}_i]\{\mathsf{DOp}_1,\ldots,\mathsf{DOp}_h\}(\mathsf{Op}_1,\ldots,\mathsf{Op}_k)$

with Op being the operator name; p_i's being the static parameters of the operator; DOp_i's being dependent sub-operators; and Op_i's are input (or independent) operators. [...] stands for tuple construction, # for tuple field access, and IN is the input of dependent sub-operators (as passed by the operator from the result of its independents sub-operators). Path expressions, constructors, and update operators are written in XQuery! syntax for conciseness.

```
Snap (
  MapFromItem{
    <person name="{ IN#p/name }">{ count(IN#a) }</person>
   }
   (GroupBy [a,index,null]
        { IN }
        { (insert { <buyer person="{IN#t/buyer/@person}"
                              itemid="{IN#t/itemref/@item}" /> }
            as last into { $purchasers }, IN#t) }
    (LOuterJoin[null]{ IN#t/buyer/@person = IN#p/@id }
         (MapIndexStep[index]
             (MapFromItem{[p:IN]}($auction//person)),
           MapFromItem{[t:IN]}($auction//closed_auction)))))
```

In general, the optimization rules must be guarded by appropriate preconditions to ensure that not only the resulting value is correct, but also that the order (when applicable) and the values of side-effects are preserved. Those preconditions check for properties related to cardinality and a notion of independence between expressions. The former ensures that expressions are evaluated with the correct cardinality, as changing the number of invocation may change the number of effects applied to the store. The latter is used to check that a part of the query cannot observe the effects resulting from another part of the query, hence allowing certain rewritings to occur.

More specifically, consider the compilation of a join from nested for loops (maps): we must check that the inner branch of a join does not have updates. If the inner branch of the join does have update operations, they would be applied once for each element of the outer loop. Merge join and hash joins are efficient because they only evaluate their inputs once, however doing so may change the cardinality for the side-effect portion of the query. Additionally, we must ensure that applying these new updates does not change the values returned in the outer branch, thus changing the value returned by the join. The first problem requires some analysis of the query plan, while the latter is difficult to ensure without the use of snap. In our example, if we had used a `snap insert` at line 5 of the source code, the group-by optimization would be more difficult to detect as one would have to know that the effect of the inserts are not observed in the rest of the query. This bears some similarity with the technique proposed in [1], although it is applied here on a much more expressive language.

5 Related work

Nested transactions. The `snap` operator groups update requests to apply them all at once, which is reminiscent of transactions. However, their purpose and semantics are essentially orthogonal. Flat transactions are meant to protect a piece of code from concurrently running transactions, while nested transactions allow the programmer to isolate different concurrent threads within its own code.

On the other side, without concurrency and failures, transactions have no effect. In particular, within a given transaction, access to a variable *x* that has been modified will return the new value for that variable. On the contrary, an update to *x* requested inside a `snap` scope will not affect the result of queries to *x* inside the same scope. In a nutshell, transactions isolate against external actions, while `snap` delays internal actions.

Monads in pure functional languages. Our approach allows the programmer to write essentially imperative code containing code fragments which are purely functional, and hence can be optimized more easily. The motivation is similar to that of monadic approaches in languages such as Haskell [13]. In those approaches, the type system distinguishes between purely functional code, which can be lazily evaluated, from impure "monadic" code, for which evaluation order is constrained. The type system will not allow pure code to call monadic code, while monadic code may invoke pure code at will.

An XQuery! processor must also distinguish the case where the query has some pending updates but no effect, going beyond the pure-impure distinction. Those pieces of code in XQuery! do not block every optimizations, provided that some "independence" constraints are verified. It seems that these constraints are too complex to be represented through types. Hence, we let the optimizer collect the relevant information, and in particular flag the scope of each innermost `snap` as *pure*. To be fair, we believe that a bit of typing would be useful: the signature of functions coming from other modules should contain an *updating* flag, with the "monadic" rule that a function that calls an updating function is *updating* as well. We are currently investigating the systematic translation of XQuery! to a core monadic language, which should give us a more complete understanding of the relationship between the two approaches.

Snapshot semantics and optimization. The optimization opportunities enabled by the snapshot semantics are explored in [1]. An important difference is that we consider similar optimization in the context of a fully compositional language.

6 Conclusion

We presented a semantic framework, and an extension of XQuery 1.0 that supports fully compositional updates. The main contribution of our work is the definition of a `snap` operator which enables control over update application and supports arbitrary nesting. We described a prototype implementation which is available for download. Many important issues are still open for research, such as static typing, optimization, and transactional mechanisms. We are currently working on those issues.

Acknowledgments. We want to thank the members of the W3C XML Query working group update task for numerous discussions on updates. Thanks to Daniela Florescu, Don Chamberlin, Ioana Manolescu, Andrew Eisenberg, Kristoffer Rose, Mukund Raghavachari, Rajesh Bordawekar, and Michael Benedikt for their feedback on earlier versions of this draft. Special thanks go to Dan Suciu for proposing `snap` as the keyword used in XQuery!.

References

1. Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P'05*, 2005.
2. Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Simeon. XQuery 1.0: An XML query language. W3C Working Draft, April 2005.
3. Don Chamberlin. Communication regarding an update proposal. W3C XML Query Update Task Force, May 2005.
4. Don Chamberlin and Jonathan Robie. XQuery update facility requirements. W3C Working Draft, June 2005.
5. Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jerôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C Working Draft, Aug 2004. `http://www.w3.org/TR/query-semantics`.
6. Daniela Florescu et al. Communication regarding an XQuery update facility. W3C XML Query Working Group, July 2005.
7. Don Chamberlin et al. Communication regarding updates for XQuery. W3C XML Query Working Group, October 2002.
8. Mary Fernández, Jerôme Siméon, and Philip Wadler. *XQuery from the experts*, chapter Introduction to the Formal Semantics. Addison Wesley, 2004.
9. Daniela Florescu. Communication regarding update grammar. W3C XML Query Update Task Force, April 2005.
10. Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: An XML programming language for Web service specification and composition. In *Proceedings of International World Wide Web Conference*, pages 65–76, May 2002.
11. Giorgio Ghelli, Christopher Ré, and Jérôme Siméon. XQuery!: An XML query language with side effects, full paper, 2005. `http://xquerybang.cs.washington.edu/papers/XQueryBangTR.pdf`.
12. Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In *Database and XML Technologies (XSym)*, pages 5–20, May 2004.
13. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In "Engineering theories of software construction", ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, 2001.
14. Patrick Lehti. Design and implementation of a data manipulation processor for an XML query processor, Technical University of Darmstadt, Germany, Diplomarbeit, 2001.
15. Xavier Leroy. *The Objective Caml system, release 3.08, Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, july 2004.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. MIT Press, 1997.
17. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. Technical report, AT&T Labs Research, 2005.
18. Christopher Ré, Jerome Simeon, and Mary Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta,GA, April 2006.
19. Michael Rys. Proposal for an XML data modification language, version 3, May 2002. Microsoft Corp., Redmond, WA.
20. A. Schmidt, F. Waas, M. Kersten, M. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, August 2002.
21. Gargi M. Sur, Joachim Hammer, and Jérôme Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

Conflict Resolution in Updates through XML views

André Prisco Vargas[1], Vanessa P. Braganholo[2], and Carlos A. Heuser[1]

[1] Instituto de Informática - Federal University of Rio Grande do Sul - Brazil
[2] COPPE - Federal University of Rio de Janeiro - Brazil
[apvargas,heuser]@inf.ufrgs.br, vanessa@cos.ufrj.br

Abstract. In this paper, we focus on B2B scenarios where XML views are extracted from relational databases and sent over the Web to another application that edits them and sends them back after a certain (usually long) period of time. In such transactions, it is unrealistic to lock the base tuples that are in the view to achieve concurrency control. Thus, there are some issues that need to be solved: first, to identify what changes were made in the view, and second, to identify and solve conflicts that may arise due to changes in the database state during the transaction. We address both of these issues in this paper by proposing an approach that uses our XML view update system PATAXÓ.

1 Introduction

XML is increasingly being used as an exchange format between business to business (B2B) applications. In this context, a very common scenario is one in which data is stored in relational databases (mainly due to the maturity of the technology) and exported in XML format [11, 8] before being sent over the Web. The proposes in [11, 8], however, address only part of the problem, that is, they know how to generate and query XML views over relational databases, but they do not know how to update those views. In B2B environments, enterprises need not only to obtain XML views, but also to update them. An example is a company *B* (buyer), that buys products from another company *S* (supplier). One could think on *B* asking *S* for an *order form*. *B* would them receive this form (an empty XML view) in a PDA of one of its employees who would fill it in and send it back to *S*. *S* would them have to process it and place the new order in its relational database. This scenario is not so complicated, since the initial XML view was empty. There are, however, more complicated cases. Consider the case where *B* changes its mind and asks *S* its order back, because it wants to change the quantities of some of the products it had ordered before. In this case, the initial XML view is not empty, and *S* needs to know what changes *B* made to it, so it can reflect the changes back to the database.

In previous work [2], we have proposed PATAXÓ, an approach to update relational databases through XML views. In this approach, XML views are constructed using UXQuery [1], an extension of XQuery, and updates are issued through a very simple update language. The scenario we address in this paper is different in the following senses: (i) In PATAXÓ [2], updates are issued through an update language that allows insertions, deletions and modifications. In this paper, we deal with updates done directly over the XML view, that is, users directly *edit* the XML view. Thus, we need to know exactly what changes were made to the view. We address this by calculating the *delta* between the original and the updated view. Algorithms in literature [5, 3, 14, 6] may be used in this case, but need to be adapted for the special features of the updatable XML views produced by PATAXÓ; (ii) In PATAXÓ [2], we rely on the transaction manager of the underlying DBMS. As most DBMS apply the ACID transaction model, this means that we simple lock the database tuples involved in a view until all the updates have been translated to the database. In B2B environments, this is impractical because the transactions may take a long time to complete [4]. Returning to our example, company *B* could take days to submit the changes to its order back to *S*. The problem in this case is what to do when the database state changes during the transaction (because of external updates). In such cases, the original XML view may not be valid anymore, and conflicts may occur.

In this paper, we propose an approach to solve the open problems listed above. We use PATAXÓ [2] to both generate the XML view and to translate the updates over the XML view back to the underlying relational database. For this to be possible, the update operations that were executed over the XML view need to be detected and specified using the PATAXÓ update language. It is important to notice that not all update operations are valid in this context. For example, PATAXÓ does not allow changing the tags of the XML elements, since this modifies the view schema – this kind of modification can not be mapped back to the underlying relational database.

We assume the XML view is updatable. This means that all updates applied to it can be successfully mapped to the underlying relational database. In [2], we present a set of rules the view definition query must obey in order for the resulting XML view to be updatable. Basically, this means that primary keys are preserved in the view, joins are made by key-foreign keys, and nesting is done from the owner relation to the owned relation. An example of non-updatable view would be a view that repeats the customer name for each item of a given order. This redundancy causes problems in updates, thus the view is not updatable.

Application Scenario Consider companies *B* and *S*, introduced above. Company *S* has a relational DB that stores orders, products and customers. The DB schema is shown in Figure 1(a). Now, let's exemplify the scenario previously described. Company *B* requests its order to company *S* so it can alter it. The result of this request is the XML view shown in Figure 2 (the numbers near the nodes, shown in red in the Figure, are used so we can refer to a specific node in our examples). While company *B* is analyzing the view and deciding what changes it will make over it, the relational database of company *S* is updated as shown in Figure 1(b). These updates may have been made directly over the database, or through some other XML view. The main point is that the update over *LineOrder* affects the XML view that is being analyzed by company *B*. Specifically, it changes the price of one of the products that *B* has ordered (blue pen).

Meanwhile, *B* is still analyzing its order (XML view) and deciding what to change. It does not have any idea that product ”blue pen” had its price doubled. After 5 hours, it decides to make the changes shown in Figure 3 (the changes are shown in boldface in the figure). The changes are: increase the quantity of blue pens to 200, increase the quantity of red pens to 300, and order a new item (100 notebooks (NTBK)).

```
                       (a)
Customer (custId, name, address),
   primary key (custId)
Product (prodId, description, curPrice),
   primary key (prodId)
Order (numOrder, date, custId, status),
   primary key (numOrder),
   foreign key (custId) references Customer
LineOrder (numOrder, prodId, quantity, price),
   primary key (numOrder, prodId),
   foreign key (prodId) references Product,
   foreign key (numOrder) references Order
```

```
                (b)
//increases price of "blue pen"
UPDATE Product
SET curPrice = 0.10
WHERE prodId = "BLUEPEN";

UPDATE LineOrder
SET price = 0.10
WHERE prodId = "BLUEPEN"
AND status="open";
```

Fig. 1. (a) Sample database of company *S* (b) Updates made over the database

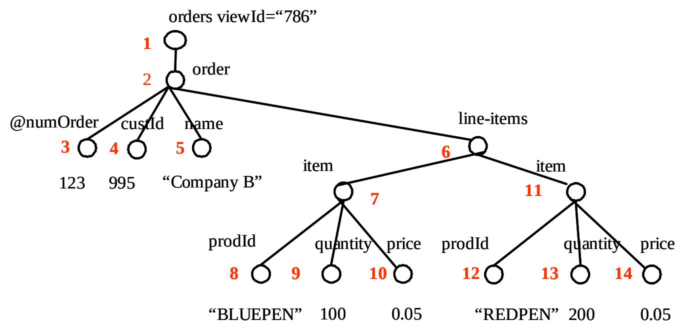


Fig. 2. Original XML view

When *S* receives the updated view, it will have to: (i) Detect what were the changes made by *B* in the XML view; (ii) Detect that the updates shown in Figure 1(b) affect the view returned by *B*, and detect exactly what are the conflicts; (iii) Decide how to solve the conflicts, and update the database using PATAXÓ.

Related Work A problem closely related to our work is the problem of consistency control of disconnected database replicas [9, 12, 10]. To solve such problem, Phatak and Badrinath [10] propose a reconciliation phase that synchronizes operations. [9] uses conflict detection and dependencies between operations. However, these approaches do not deal with the XML views problem or require the semantics of the data to be known. In our paper, we use some of the ideas of [9] in the XML context. In [13], we provide more details on related work.

Contributions and Organization of the Text The main contributions of this paper are: (i) A delta detection technique tailored to the PATAXÓ XML views; (ii) An approach to verify the status of the database during the transaction. This is done by comparing the status of the database in the beginning of the transaction with the status of the database in the time the updated view is returned to the system; (iii) A conflict resolution technique, based on the structure of the XML view; (iv) A merge algorithm to XML views that emphasizes the conflicts caused by changes in the database state during the transaction.

The remaining of this paper is organized as follows. Section 2 presents an overview of the PATAXÓ approach. Section 3.1 presents our technique to detect deltas in XML

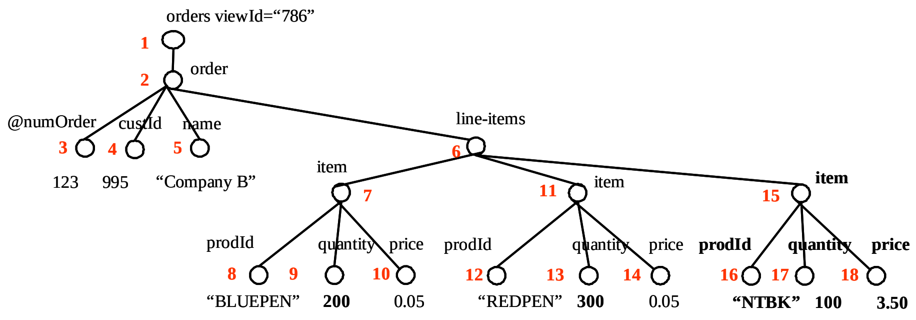


Fig. 3. XML view updated by company *B* and returned to company *S*

views, and Section 3.2 presents a solution to the problems caused by conflicts. Finally, we conclude in Section 4.

2 The PATAXÓ approach

As mentioned before, PATAXÓ [2] is a system that is capable of constructing XML views over relational databases and mapping updates specified over this view back into the underlying relational database. To do so, it uses an existing approach on updates through relational views [7]. Basically, a view query definition expressed in UXQuery [1] is internally mapped to a query tree [2]. Query trees are a formalism that captures the structure and the source of each XML element/attribute of the XML view, together with the restrictions applied to build the view. As an example, the query tree that corresponds to the view query that generates the XML view of Figure 2 is shown if Figure 4. The interested reader can refer to [2] for a full specification of query trees.

In this paper, it will be important to recognize certain types of nodes in the query tree and in the corresponding view instance. In the query tree of Figure 4, node *order* is a *starred-node* (*-node). Each starred node generates a collection of (possibly complex) elements. Each such element carries data from a database tuple, or from a join between two or more tuples (tables Customer and Order, in the example). We call each element of this collection a *starred subtree*. The element itself (the root of the subtree), is called *starred element*. In the example of Figure 2 (which is generated from the query tree of Figure 4), nodes 2, 7 and 11 are *starred elements* (since they are produced by starred nodes of the corresponding query tree).

Updates in PATAXÓ As mentioned before, PATAXÓ uses a very simple update language. Basically, it is expressed by a triple $\langle t, \Delta, ref \rangle$, where t is the type of the operation (*insert*, *delete* or *update*), Δ is the subtree to be inserted or an atomic value to be modified, and *ref* is a path expression that points to the update point in the XML view. The update point *ref* is expressed by a simple XPath expression that only contains child access (/) and conjunctive filters.

Not all update specifications are valid, since they need to be mapped back to the underlying relational database. Mainly, the updates applied to the view need to follow

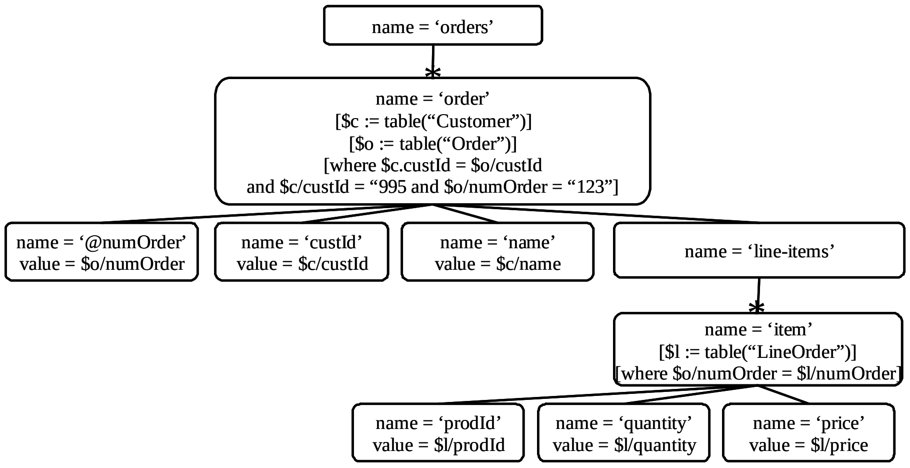


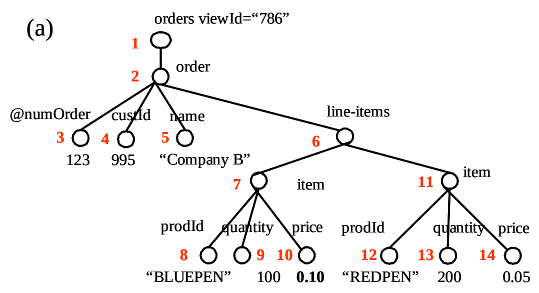
Fig. 4. Query tree that generated the XML view of Figure 2

the view DTD. PATAXÓ generates the view together with its DTD, and both the view and the DTD are sent to the client application. The DTD of the XML view of Figure 2 is available in [13]. The remaining restrictions regarding updates are as follows: (i) subtrees inserted must represent a (possibly complex/nested) database tuple. This restriction corresponds to adding only subtrees rooted at starred nodes in the query trees of [2]. Such elements correspond to elements with cardinality ”*” in the DTD. Thus, in this paper, it is enough to know that only subtrees rooted at elements with cardinality ”*” in the DTD can be inserted. In Figure 3 the inserted subtree `item` (node 15) satisfies this condition. (ii) The above restriction is the same for deletions. Subtrees deleted must be rooted at a starred node in the query tree. This means that in the example view, we could delete `order` and `item` subtrees.

All of these restrictions can be verified by checking the updated XML view against the DTD of the original view. As an example, it would not be possible to delete node `name` (which is not a starred element, and so contradicts rule (ii) above), since this is a required element in the DTD. Also, it is necessary to check that updates, insertions and deletions satisfy the view definition query. As an example, it would not be possible to insert another `order` element in the view, since the view definition requires that this view has only an order with `numOder` equals ”123” (see the restrictions on node *order* of Figure 4).

3 Supporting Disconnected Transactions

In this section, we describe our approach and illustrate it using the order example of Section 1. Our architecture [13] has three main modules: the *Transaction Manager*, *Diff Finder* and *Update Manager*. The *Transaction Manager* is responsible for controlling the currently opened transactions of the system. It receives a view definition query, passes it to PATAXÓ, receives PATAXÓ’s answer (the resulting XML view and its



(b)
$E_1(O \rightarrow U)$ = Update(9, 200), Update(13, 300), Insert(t_1, 6))

```
t1 = <item>
       <prodId>NTBK</prodId>
       <quantity>100</quatity>
       <price>3.50</price>
     </item>
```

$E_2(O \rightarrow view')$ = Update(10, 0.10)

Fig. 5. (a) View' (b) Edit scripts for our example

DTD), and before sending it to the client, it: (i) adds an `viewId` to the root of the XML view (this attribute is set to 786 in the example view of Figure 2; the value that will be assigned to attribute `viewId` is controlled by a sequential counter in the Transaction Manager); (ii) adds this same attribute, with the same value, to the root of the view definition query; (iii) adds and attribute declaration in the view DTD for the `viewId`; (iv) stores the XML view, the view definition query and the view DTD in a *Temporary Storage* facility, since they will have to be used later when the updated view is returned to the system.

When the updated view is returned by the user to the system, the Transaction Manager checks its `viewId` (it is a requirement of our approach that the `viewID` is not modified during the transaction) and uses it to find the view DTD and the definition query, which are stored in the Temporary Storage facility. Then it uses the DTD to validate the updated view. If the view is not valid, then the transaction is aborted and the user is notified. In the case it is valid, then the Transaction Manager sends the view definition query to PATAXÓ, and receives a new XML view reflecting the database state at this moment as a response (we will call it *view'*). This new XML view will be used to check the database state. If it is exactly the same as the original XML view (which is also stored in the temporary storage facility), then the updates made to the database during this transaction do not affect the XML view. In this case all view updates made in the updated XML view may be translated back to the database. Notice that, at this stage, we have three copies of the XML view in the system:

– The *original* XML view (*O*): the view that was sent to the client at the first place. In our example, the original view is shown in Figure 2.
– The *updated* XML view (*U*): the view that was updated by the client and returned to the system. The updated XML view is shown in Figure 3.
– The *view'*: a new XML view which is the result of running the view definition query again, right after the updated view arrives in the system. *View'* is used to capture possible conflicts caused by external updates in the base tuples that are in the original view. As an example, view' is shown in Figure 5(a). Notice that it reflects the base updates shown in Figure 1(b).

These views are sent to the *Diff Finder* module. In this module, two comparisons take place. First, the *original* view is compared to the *updated* view, to find what were the updates made over the view. Next, the *original* view is compared to *view'* to find out if the database state has changed during the transaction (Section 3.1). The deltas found by *Diff Finder* are sent to the *Update Manager*, which analyzes them and detects conflicts. In case there are no conflicts, the Update Manager transforms the updates into updates using the PATAXÓ update language and sends them to PATAXÓ. PATAXÓ then translates them to the relational database. If there are conflicts we try to solve them (Section 3.2).

3.1 Detecting deltas in XML views

As mentioned before, the *Diff Finder* is responsible for detecting the changes made in the XML view, and also in the database (through the comparison of *view'* with the original view). To accomplish this, it makes use of an existing diff algorithm that finds *deltas* between two XML views. A *delta* is a set of operations that denotes the difference between two data structures D_1 and D_2 in a way that if we apply *delta* to D_1, we obtain D_2. Using the notation of [14], this delta can be expressed by $E(D_1 \rightarrow D_2)$.

We adopt X-Diff [14] as our diff algorithm, mainly because it is capable of detecting the operations supported by PATAXÓ (insertion of subtrees, deletion of subtrees and modification of text values), and considers an unordered model. MH-DIFF [3] does not support insertion and deletion of subtrees, (it supports only insertion and deletion of single nodes), thus it is not appropriate in our context. Xy-Diff [5] and XMLTreeDiff [6] consider ordered models, which does not match the unordered model of our source data (relations).

X-DIFF. According to [14], the operations detected by X-Diff are as follows:

Insertion of leaf node The operation *Insert(x(name, value), y)* inserts a leaf node x with name *name* and value *value*. Node x is inserted as a child of y.

Deletion of leaf node Operation *Delete(x)* deletes a leaf node x.

Modification of leaf value A modification of a leaf value is expressed as *Update(x, new-value)*, and it changes the value of node x to *new-value*.

Insertion of subtree Operation *Insert*(T_x, *y*) inserts a subtree T_x (rooted at node x) as a child of node y.

Deletion of subtree The operation *Delete*(T_x) deletes a subtree T_x (rooted at node x). When there is no doubts about which is x, this operation can be expressed as *Delete(x)*.

An important characteristic of X-Diff is that it uses parent-child relationships to calculate the minimum-cost matching between two trees T_1 and T_2. This parent-child relationship is captured by the use of a node *signature* and also by a hash function. The hash function applied to node y considers its entire subtree. Thus, two equal subtrees in T_1 and T_2 have the same hash value. The node signature of a node x is expressed by $Name(x_1)/.../Name(x_n)/Name(x)/Type(x)$, where ($x_1/.../x_n/x$) is the path from the root to x, and *Type(x)* is the type of node x. In case x is not an atomic element, its signature does not include *Type(x)* [14]. Matches are made in a way that only nodes with the same signature are matched. Also, nodes with the same hash value are identical subtrees, and thus they are matched by X-Diff.

To exemplify, Figure 5(b) shows the edit script generated by X-Diff for the original (O) and updated (U) views. This Figure also shows the edit script for the original (O) view and view', which is also calculated by *Diff Finder*.

Update Manager The Update Manager takes the edit script generated by X-Diff and produces a set of update operations in the PATAXÓ update language. Here, there are some issues that need to be taken care of. The main one regards the *update path expressions* (they are referred to as *ref* in the update specification). In PATAXÓ, update operations need to specify an update path, and those are not provided by the edit script generated by X-Diff.

To generate the update path *ref*, we use the DB primary keys as filters in the path expression. Notice that keys must be kept in the view for it to be updatable [2]. Specifically, for an operation on node x, we take the path p from x to the view root, and find all the keys that are descendants of nodes in p.

In our example, the keys are *custId, numOrder* and *prodId*. The rules for translating an X-Diff operation into a PATAXÓ operation are as follows. The function *generateRef* uses the primary keys to construct filters, as mentioned above. The general form of a PATAXÓ update operation is $\langle t, \Delta, ref\rangle$.

- *Insert (x (name, value), y)* is mapped to $\langle insert, x, generateRef\,(y)\rangle$.
- *Delete (x)* is mapped to $\langle delete, \{\}, generateRef\,(x)\rangle$.
- *Update (x, new-value)* is mapped to $\langle modify, \{new\text{-}value\}, generateRef\,(x)\rangle$.
- *Insert (T_x, y)* is mapped to $\langle insert, T_x, generateRef\,(y)\rangle$.
- *Delete (T_x)* is mapped to $\langle delete, \{\}, generateRef\,(x)\rangle$.

Function *generateRef*(x) works as follows. First, it gets the parent x_n of x, then the parent x_{n-1} of x_n, and continues to get their parents until the root is reached. The obtained elements form a path $p = x_1/.../x_{n-1}/x_n/x$. Then, for each node y in p, it searches for leaf children that are primary keys in the relational database. Use this set of nodes to specify a conjunctive filter that uses the node name and its value in the view. As an example, we show the translation of an operation of E_1 (Figure 5(b)):

- *Update(9, 200)* $\equiv$ *<modify*, $\{200\}$*, orders/order[@numOrder="123" and custId="995"]/line-item/item[prodId="BLUEPEN"]/quantity>*

PATAXÓ uses the values in the filters in the translation of modifications and deletions, and the values of leaf nodes in the path from the update point to the root in the translation of insertions. This approach, however, causes a problem when some of these values were modified by the user in the view. To solve this, we need to establish an order for the updates. This order must make sure that if an update operation u references a node value x that was modified in the view, then the update operation that modifies x must be issued *before u*. Given this scenario, we establish the following order for the updates: (1) Modifications; (2) Insertions; (3) Deletions.

There is no possibility of deleting a subtree that was previously inserted, since this kind of operation would not be generated by X-Diff. When there is more than one update in each category, then the updates that have the shortest update path (*ref*) are issued first. To illustrate, consider a case where the numOrder is changed (u_1), and the quantity of an item is changed by u_2. Since the numOrder is referenced in the filter of the update path of u_2, then u_1 has to be issued first, so that when u_2 is executed, the database already has the correct value of the numOrder. Notice that this example is not very common in practice, since normally primary key values are not changed.

3.2 Guaranteeing database consistency

The detection of conflicts is difficult, because a conflict can have different impacts depending on the application. To illustrate, in our example of orders, the removal of a product from the database means that the customer can not order it anymore. As a counter example, if a user increases the quantity of an item in its order, she may not want to proceed with this increase when she knows that the price of the item has increased.

The issues above are semantic issues. Unfortunately, a generic system does not know about these issues, and so we take the following approach: The *Diff Finder* uses X-Diff to calculate the edit script for the original XML view *O* and the view that has the current database state (*view'*). If the edit script is empty the updates over the updated view can be translated to the database with no conflict. In this case, the Update Manager translates the updates to updates in the PATAXÓ update language (Section 3.1) and sends them to PATAXÓ so it can map them to the underlying relational database.

However, most of the times the views (*O* and view') will not be equal, which implies in conflicts. A conflict is any update operation that has been issued in the database during the transaction lifetime, and that affects the updates made by the user through the view. We will provide more details on this later on.

In our approach, there are two *operational modes* to deal with conflicts. The first one is the *restrictive mode*, in which no updates are translated when there are differences between the views original and view'. This is a very restrictive approach, where all modifications made over the view are treated as a single atomic transaction. The second, less restrictive mode of operation is called *relaxed mode*. In this mode, updates that do not cause conflicts are translated to the underlying database. The remaining ones are aborted. To keep database consistency, we assume that some updates may coexist with others done externally, without causing inconsistencies in the database. To recognize such cases, we define a set of rules that are based on the view structure only. Notice that we do not know the semantics of the data in the view nor in the database. Thus, sometimes we may detect an operation to cause conflict even tough semantically it does not cause conflicts. This is the price we pay for not requiring the user to inform the semantics of the data in the view.

Conflict Detection Rules In this section, we present rules for the resolution of conflicts in modifications. We leave insertions and deletions for future work.

Rule 1 *(Leaf node within the same starred-element) Let* $L = \{l_1, ..., l_n\}$ *(*$n \geq 1$*)) be the set of leaf nodes descending from a starred node s in a given XML view v. Additionally, ensure that s is the first starred ancestor of the nodes in L. If any* $l_i \in L$ *is modified in the updated view, and some* l_j *is modified in view' (*$i = j$ *or* $i \neq j$*), then the updates in nodes of L are rejected.*

An example of such case can be seen in the modification of node 9 (quantity of blue pens) in Figure 3 from 100 to 200. This operation can not proceed because it conflicts with the update of node 10 (price of blue pens) in view'.

Rule 2 *(Dependant starred-subtrees) Let* s_1 *and* s_2 *be two starred subtrees in a given XML view v. Let* $L_1 = \{l_{1_1}, ..., l_{1_n}\}$ *(*$n \geq 1$*)) be the set of leaf nodes descending from* s_1*,* *but not from its starred subtrees, and* $L_2 = \{l_{2_1}, \ldots, l_{2_k}\}$ *(*$k \geq 1$*)) be the set of leaf nodes descending from* s_2*, but not from its starred subtrees. Further, let* s_1 *be an ancestor of* s_2*. If any* $l_{2_i} \in L_2$ *is modified in the updated view, and some* $l_{1_j} \in L_1$ *is modified in view', then the updates conflict, and the modification of* l_{2_i} *is aborted.*

This rule captures the dependency between starred subtrees. In the XML view of Figure 3, it is easy to see that each *item* subtree is semantically connected to its parent *order* tree. Thus, rule 2 defines that modifications done in the database that affect the *order* subtree conflicts with modifications to the *item* subtree done through the view.

Notice that in all the above rules, we need to know the correspondence of nodes in views *U* and *view'*. For example, we need to know that node 12 in the updated view (Figure 3) correspond to node 12 in *view'* (Figure 5(a)). This can be easily done by using a variation of our *merge* algorithm presented later on.

To check for conflicts, each modify operation detected in $E_1(O \to U)$ is checked against each modify operation in $E_2(O \to \textit{view'})$ using the rules above. In [13], we present the algorithm. The checking is very simple, and once we detect a conflict due to one rule, we do not need to check the other one.

Notifying the User In both configuration modes of our system, we need to inform the user of which update operations were actually translated to the base tables, and which were aborted. To do so, the system generates a *merge* of the updated data and the current database state. The algorithm starts with the original XML view. Consider $E1 = E(O \to U)$ and $E2 = E(O \to \textit{view'})$.

1. Take each delete operation u=Delete(x) in $E2$ and mark x in the original XML view. The markup is made by adding a new parent *pataxo:DB-DELETE* to x, where *pataxo* is a namespace prefix (we ommit it in the remaing definitions). This new element is connected to the parent of x.
2. Take each insert operation u=Insert(T_x, y) in $E2$, insert T_x under y and add a new parent *DB-INSERT* to T_x. Connect the new element as a child of y.
3. Take each modify operation u=Update(x, new-value) in $E2$, add a *DB-MODIFY* element with value *new-value*. Connect it as a child of x.

After this, it is necessary to apply the update operations that are in the updated view to the original view, and mark them too. In this step, the markup elements receive a STATUS attribute to describe if the update operation was accepted or aborted. Since we are currently detecting conflicts only between modify operations, we are assuming the remaining ones are always accepted.

1. Take each delete operation u=Delete(x) in $E1$, add a new parent *CLIENT-DELETE STATUS="ACCEPT"* to x and connect it to the parent of x.
2. Take each insert operation u=Insert(T_x, y) in $E1$, insert T_x under y and add a new parent *CLIENT-INSERT STATUS="ACCEPT"* to T_x. Connect the new created element as a child of y.
3. Take each modify operation u=Update(x, new-value) in $E1$, add a new element *CLIENT-MODIFY* with value *new-value*. Connect the *CLIENT-MODIFY* element as a child of x. If u is marked in $E1$, then add a *STATUS* attribute to the *CLIENT-MODIFY* with value *ABORT*. If not, then add the *STATUS* attribute with value *ACCEPT*.

```
<orders viewId="786">
   <order numOrder="123">
      <custId>995</custId>
      <name>Company B</name>
      <line-items>
         <item>
            <prodId>BLUEPEN</prodId>
            <quantity>100
                <pataxo:CLIENT-MODIFY STATUS="ABORT">
                200
                </pataxo:CLIENT-MODIFY>
            </quantity>
            <price>0.05<pataxo:DB-MODIFY>0.10</pataxo:DB-MODIFY></price>
         </item>
         <item>
            <prodId>REDPEN</prodId>
            <quantity>200
                <pataxo:CLIENT-MODIFY STATUS="ACCEPT">
                300
                <pataxo:CLIENT-MODIFY>
            </quatity>
            <price>0.05</price>
         </item>
         <pataxo:CLIENT-INSERT STATUS="COMMIT">
            <item>
              <prodId>NTBK</prodId>
              <quantity>100</quantity>
              <price>3.50</price>
            </item>
         </pataxo:CLIENT-INSERT>
      </line-items>
   </order>
</orders>
```

Fig. 6. Result of the *merge* algorithm

The result of this merge in our example is shown in Figure 6. There may be elements with more than one conflict markup. For example, suppose the client had altered the price of blue pens to 0.02 (the issue of whether this is allowed by the application or not, is out of the scope of this paper). In this case, the element price would have two markups.

After the execution of the merge algorithm, the Transaction Manager receives the new *merged* view (notice that the merged view is an XML document not valid according to the view DTD, since new markup elements were added). It re-generates the view (which is now the original view *O*), and stores it in the temporary storage facility, since now this is the new *original view*. Then, it sends the *merged* view and view *O* back to the client application. The client may want to analyze the merged view and to resubmit updates through view *O*. This second ”round” will follow the same execution flow as before. The system will proceed as if it was the first time that updated view arrives in the system.

4 Discussion and Future Work

We have presented an approach to support disconnected transactions in updates over relational databases through XML views. Our approach uses PATAXÓ [2] to both generate the views and to translate the updates to the underlying relational database. In this paper, we allow views to be edited, and we automatically detect the changes using X-Diff [14]. We present an algorithm to transform the changes detected by X-Diff into the update language accepted by PATAXÓ. Also, we present a technique to detect conflicts that may be caused by updates over the base relations during the transaction execution. Currently, we only detect conflicts for modifications.

One of the benefits of our approach is that it does not require that updates are done online. In our previous approach [2], the client application must be connected to PATAXÓ in order to issue updates. In this paper, however, we support offline update operations that can be done in offline devices, like PDAs.

This scenario is very common in practice, and we believe that industry will greatly benefit from our work. In the future, we plan to evaluate our approach in real enterprises. Also, we are working on rules to detect conflicts on insertions and deletions. We plan to work on algorithms to solve such conflicts.

References

1. V. Braganholo, S. Davidson, and C. Heuser. UXQuery: building updatable XML views over relational databases. In *SBBD*, pages 26–40, Brazil, 2003.
2. V. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, Toronto, Canada, Sept. 2004.
3. S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, pages 26–37, Tucson, Arizona, May 1997.
4. P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems, TODS*, 19(3):450–491, 1994.
5. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE*, pages 41–52, San Jose, California, Feb. 2002.
6. F. Curbera and D. Epstein. Fast difference and update of XML documents. In *XTech*, San Jose, California, Mar. 1999.
7. U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 8(2):381–416, Sept. 1982.
8. M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in XML. *ACM TODS*, 27(4):438–493, Dec. 2002.
9. L. Klieb. Distributed disconnected databases. In *Symposium on Applied Computing (SAC)*, pages 322–326, New York, NY, USA, 1996. ACM Press.
10. S. H. Phatak and B. R. Badrinath. Conflict resolution and reconciliation in disconnected databases. In *DEXA*, 1999.
11. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, Rome, Sept. 2001.
12. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.
13. A. Vargas, V. Braganholo, and C. Heuser. Conflict resolution and difference detection in updates through XML views. Technical Report RP-352, UFRGS, Brazil, Dec. 2005. Available at `www.cos.ufrj.br/~vanessa`.
14. Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for XML documents. In *ICDE*, pages 519–530, India, March 2003.

Efficient Integrity Checking over XML Documents

Daniele Braga[1], Alessandro Campi[1], and Davide Martinenghi[2]

[1] Politecnico di Milano – Dip. di Elettronica e Informazione
p.zza L. da Vinci 32, 20133 Milano, Italy
{braga,campi}@elet.polimi.it
[2] Free University of Bozen/Bolzano – Faculty of Computer Science
p.zza Domenicani, 3, 39100 Bolzano, Italy
martinenghi@inf.unibz.it

Abstract. The need for incremental constraint maintenance within collections of semi-structured documents has been ever increasing in the last years due to the widespread diffusion of XML. This problem is addressed here by adapting to the XML data model some constraint verification techniques known in the context of deductive databases. Our approach allows the declarative specification of constraints as well as their optimization w.r.t. given update patterns. Such optimized constraints are automatically translated into equivalent XQuery expressions in order to avoid illegal updates. This automatic process guarantees an efficient integrity checking that combines the advantages of declarativity with incrementality and early detection of inconsistencies.

1 Introduction

It is well-known that expressing, verifying and automatically enforcing data correctness is a difficult task as well as a pressing need in any data management context. In this respect, XML is no exception; moreover, there is no standard means of specifying generic constraints over large XML document collections. XML Schema offers a rich set of predefined constraints, such as structural, domain and cardinality constraints. However, it lacks full extensibility, as it is not possible to express general integrity requirements in the same way as SQL assertions, typically used to specify business rules at the application level in a declarative way. A large body of research, starting from [16], gave rise to a number of methods for incremental integrity checking within the framework of deductive databases and w.r.t. the relational data model. Indeed, a brute force approach to integrity checking, i.e., verifying the whole database each time data are updated, is unfeasible. This paper addresses this problem in the context of semi-structured data, and namely XML, in order to tackle the difficulties inherent in its hierarchical data model. A suitable formalism for the declarative specification of integrity constraints over XML data is therefore required in order to apply optimization techniques similar to those developed for the relational world. More specifically, we adopt for this purpose a formalism called XPathLog, a logical language inspired by Datalog and defined in [13]. In our approach, the tree structure of XPathLog constraints is mapped to a relational representation (in Datalog) which lends itself well to the above mentioned optimization techniques. The optimization only needs to take place once, at schema design time: it takes as input a set of constraints and an update pattern and, using the hypothesis that the database is always consistent prior to the update, it produces as output a set of optimized constraints, which are as instantiated as possible. These optimized constraints are finally translated into XQuery expressions that can be matched against the XML document so as to check that the update does not introduce any violation of the constraints.

In particular, the constraint simplification method we adopt generates optimized constraints that can be tested *before* the execution of an update (and without simulating the updated state), so that inconsistent database states are completely avoided.

2 Constraint verification

Semantic information in databases is typically represented in the form of integrity constraints, which are properties that must always be satisfied for the data to be considered *consistent*. In this respect, database management systems should provide means to automatically verify, in an efficient way, that database updates do not introduce any violation of integrity. A complete check of generic constraints is too costly in any nontrivial case; in view of this, verification of integrity constraints can be rendered more efficient by deriving specialized checks that are easier to execute at each update. Even better performance is achieved if these checks can be tested before illegal updates. Nevertheless, the common practice is still based on *ad hoc* techniques: domain experts hand-code tests in the application program producing the update requests or design triggers within the database management system that respond to certain update actions. However, both methods are prone to errors and little flexibility w.r.t. changes in the schema or design of the database, which motivates the need for automated integrity verification methods.

In order to formalize the notion of consistency, and thus the constraint verification problem, we refer to deductive databases, in which a *database state* is the set of database facts and rules (tuples and views). As semantics of a database state D we take its *standard model*: the truth value of a closed formula F, relative to D, is defined as its valuation in the standard model and denoted $D(F)$.

Definition 1 (Consistency). *A database state D is* consistent *with a set of integrity constraints Γ iff $D(\Gamma) = true$.*

An *update* U is a mapping $U : \mathscr{D} \mapsto \mathscr{D}$, where $\mathscr{D}$ is the space of database states. For convenience, for any database state D, we indicate the state arising after update U as D^U. The constraint verification problem may be formulated as follows. Given a database state D, a set of integrity constraints Γ, such that $D(\Gamma) = true$, and an update U, does $D^U(\Gamma) = true$ hold too? As mentioned, evaluating $D^U(\Gamma)$ may be too expensive, so a suitable reformulation of the problem can be given in the following terms: is there a set of integrity constraints Γ^U such that $D^U(\Gamma) = D(\Gamma^U)$ and Γ^U is easier to evaluate than Γ? In other words, the looked for condition Γ^U should specialize the original Γ, as specific information coming from U is available, and avoid redundant checks by exploiting the fact that $D(\Gamma) = true$. We observe that reasoning about the future database state D^U with a condition (Γ^U) that is tested in the present state D, complies with the semantics of *deferred* integrity checking (i.e., integrity constraints do *not* have to hold in intermediate transaction states).

Consistency requirements for XML data are not different from those holding for relational data, and constraint definition and enforcement are expected to become fundamental aspects of XML data management. Fixed-format structural integrity constraints can already be defined by using XML Schema. However, a generic constraint definition language for XML is still not present. Moreover, generic mechanisms for constraint enforcement are also lacking. In this paper we cover both aspects.

Our approach moves from a recently proposed adaptation of the framework of deductive databases to the world of semi-structured data. More precisely, we refer to

XPathLog [13] as the language for specifying generic XML constraints, which are expressed in terms of queries that must have an empty result. XPathLog is an extension of XPath modeled on Datalog. In particular, the XPath language is extended with variable bindings and is embedded into first-order logic to form XPath-Logic; XPathLog is then the Horn fragment of XPath-Logic.

Even though, in principle, we could write denials in XQuery, a declarative, first-order logic language is closer to what is usually done for relational data [9]; a logical approach leads to cleaner constraint definitions, and the direct mapping from XPathLog to Datalog helps the optimization process.

Examples. Consider two documents: `pub.xml` containing a collection of published articles and `rev.xml` containing information on reviewer/paper assignment for all tracks of a given conference. The DTDs are as follows.

```
<!-- pub.xml -->             <!-- rev.xml -->            <!ELEMENT sub(title,auts+)>
<!ELEMENT dblp (pub)*>       <!ELEMENT review (track)+>  <!ELEMENT title (#PCDATA)>
<!ELEMENT pub (title,aut+)>  <!ELEMENT track (name,rev+)><!ELEMENT auts (name)>
<!ELEMENT title (#PCDATA)>   <!ELEMENT name (#PCDATA)>
<!ELEMENT aut (name)>        <!ELEMENT rev (name,sub+)>
<!ELEMENT name (#PCDATA)>
```

Example 1. Consider the following integrity constraint, which imposes the absence of *conflict of interests* in the submission review process (i.e., no one can review papers written by a coauthor or by him/herself):

$$\begin{array}{l}\leftsquigarrow //rev[name/text() \to R]/sub/auts/name/text() \to A\\ \quad \wedge(A = R \vee //pub[aut/name/text() \to A \wedge aut/name/text() \to R])\end{array}$$

The $\mathsf{text}()$ function refers to the text content of the enclosing element. The condition in the body of this constraint indicates that there is a reviewer named R who is assigned a submission whose author has name A and, in turn, either A and R are the same or two authors of a same publication have names A and R, respectively.

Example 2. Consider a conference policy imposing that a reviewer involved in three or more tracks cannot review more than 10 papers. This is expressed as follows:

$$\begin{array}{l}\leftsquigarrow \mathsf{Cnt}_{\mathsf{D}}\{[R]; //track[/rev/name/\mathsf{text}() \to R]\} \geq 3 \wedge\\ \quad \mathsf{Cnt}_{\mathsf{D}}\{[R]; //rev[/name/\mathsf{text}() \to R]/sub\} \geq 10\end{array}$$

3 Mapping XML constraints to the relational data model

In order to apply our simplification framework to XML constraints, as will be described in Section 4, schemata, update patterns, and constraints need to be mapped from the XML domain to the relational model. Note that these mappings take place statically and thus do not affect runtime performance.

3.1 Mapping of the schema and of update statements

The problem of representing XML data in relations was considered, e.g., in [19]. Our approach is targeted to deductive databases: each node type is mapped to a corresponding predicate. The first three attributes of all predicates respectively represent, for each XML item: its (unique) node identifier, its position and the node identifier of its parent node. It is worth noting that the second attribute is crucial, as the XML data model is ordered. Whenever a parent-child relationship within a DTD is a one-to-one correspondence (or an optional inclusion), a more compact form is possible, because a new predicate for the child node is not necessary: the attributes of the child may be equivalently represented within the predicate that corresponds to the parent (allowing null values in case of optional child nodes). The documents of the previous section map to the relational schema

```
pub(Id,Pos,IdParent_{dblp},Title)         aut(Id,Pos,IdParent_{pub},Name)
track(Id,Pos,IdParent_{review},Name)      rev(Id,Pos,IdParent_{track},Name)
sub(Id,Pos,IdParent_{rev},Title)          auts(Id,Pos,IdParent_{sub},Name)
```

where *Id*, *Pos* and $IdParent_{tagname}$ preserve the hierarchy of the documents and where the PCDATA content of the `name` and `title` node types is systematically embedded into the container nodes, so as to reduce the number of predicates.

As already mentioned, mapping a hierarchical ordered structure to a flat unordered data model forces the exposition of information that is typically hidden within XML repositories, such as the order of the sub-nodes of a given node and unique node identifiers. The root nodes of the documents (`dblp` and `review`) are not represented as predicates, as they have no local attributes but only subelements; however, such nodes are referenced in the database as values for the IdParent_{dblp} and IdParent_{review} attributes respectively, within the representation of their child nodes. Publications map to the *pub* predicate, authors in `pub.xml` map to *aut*, while authors in `rev.xml` map to *auts*, and so on, with predicates corresponding to tagnames. Last, `name`s and `title`s map to attributes within the predicates corresponding to their containers.

Data mapping criteria influence update mapping. We use XUpdate [8], but other formalisms would also apply. Consider the following statement:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/review/track[2]/rev[5]/sub[6]" >
    <xupdate:element name="sub">
      <title> Taming Web Services </title> <auts> <name> Jack </name> </auts>
    </xupdate:element> </xupdate:insert-after> </xupdate:modifications>
```

In the corresponding relational model, this update statement corresponds to adding $\{\ sub(id_s,\ 7, id_r,$ "Taming Web Services"$),\ auts(id_a,\ 2, id_s,$ "Jack"$)\ \}$ where id_a and id_s represent the identifiers that are to be associated to the new nodes and id_r is the identifier associated to the target `rev` element. Their value is immaterial to the semantics of the update, provided that a mechanism to impose their uniqueness is available. On the other hand, the actual value of id_r depends on the dataset and needs to be retrieved by interpreting the `select` clause of the XUpdate statement. Namely, id_r is the identifier for the fifth (`reviewer`) child of the second (`track`) node, in turn contained into the root (`review`) node of the document `rev.xml`. Positions (7 and 2 in the second argument of both predicates) are also derived by parsing the update statement: 7 is determined as the successor of 6, according to the `insert-after` semantics of the update; 2 is due to the ordering, since the `auts` comes after the `title` element. Finally, note that the *same* value id_s occurs both as the first argument of $sub()$ and the third argument of $auts()$, since the latter represents a subelement of the former.

3.2 Mapping of integrity constraints

The last step in the mapping from XML to the framework of deductive databases is to compile denials into Datalog. We express constraints as Datalog denials: clauses with an empty head (understood as *false*), whose body indicates not holding conditions. Input to this phase are the schemata (XML and relational) and an XPathLog denial in a normal form without disjunctions (a rewriting allows one to reduce to such form all denials). All p.e. in XPathLog generate chains of conditions over the predicates corresponding to the traversed node types. Containment translates to correspondences between variables in the first position of container and in the third of contained item. XPathLog denial expressing that the author of the "Duckburg tales" cannot be Goofy and its mapping (anonymous variables are indicated with an underscore):

$$\leftsquigarrow //pub[title = \text{“}Duckburg\ tales''\text{”}]/aut/name \rightarrow N \wedge\ N = \text{“}Goofy''$$

$$\leftarrow pub(I_p, \_, \_, \text{“}Duckburg\ tales''\text{”}) \wedge aut(\_, \_, I_p, N) \wedge\ N = \text{“}Goofy''.$$

The fact that the XML data model is ordered impacts the translation. Either the `position()` function is used in the original denial or a filter is used that contains an expression returning an integer. In both cases, the second argument in the relational predicate is associated to a variable that is matched against a suitable comparison expression (containing the variable associated to the `position()` function or directly to the expression that returns the value).

Example 3. The XPathLog constraint of example 1, is translated into the following couple of Datalog denials (due to the presence of a disjunction).

$$\begin{aligned} \Gamma = \{ &\leftarrow rev(I_r, \_, \_, R) \wedge sub(I_s, \_, I_r, \_) \wedge auts(\_, \_, I_s, R), \\ &\leftarrow rev(I_r, \_, \_, R) \wedge sub(I_s, \_, I_r, \_) \wedge auts(\_, \_, I_s, A) \\ &\quad \wedge aut(\_, \_, I_p, R) \wedge aut(\_, \_, I_p, A) \} \end{aligned}$$

4 Simplification of integrity constraints

Several methods for optimized and incremental constraint checking in deductive databases, known as *simplification* methods, were produced since the landmark contribution by Nicolas [16]. Simplification in this context means to derive specialized versions of the integrity constraints w.r.t. given update patterns, employing the hypothesis that the database is initially consistent. In the following, we briefly describe the approach of [11]. To illustrate the framework, we limit our attention to tuple insertions, consistently with the fact that XML documents typically grow. An update transaction is expressed as a set of ground atoms representing the tuples that will be added to the database. Placeholders for constants, called *parameters* (written in boldface: **a**, **b**, ...), allow one to indicate update *patterns*. For example, the notation $\{p(\mathbf{a}), q(\mathbf{a})\}$, where **a** is a parameter, refers to the class of update transactions that add the same tuple to both unary relation p and unary relation q. The first step in the simplification process is to introduce a syntactic transformation After that translates a set of denials Γ referring to the updated database state into another set Σ that holds in the present state if and only if Γ holds after the update.

Definition 2. *Let Γ be a set of denials and U an update. The notation $\mathsf{After}^U(\Gamma)$ refers to a copy of Γ in which all atoms of the form $p(\vec{t})$ have been simultaneously replaced by $(p(\vec{t}) \vee \vec{t} = \vec{a}_1 \vee \cdots \vee \vec{t} = \vec{a}_n)$, where $p(\vec{a}_1), \ldots, p(\vec{a}_n)$ are all additions on p in U, $\vec{t}$ is a sequence of terms and $\vec{a}_1, \ldots, \vec{a}_n$ are sequences of constants or parameters (we assume that the result of this transformation is always given as a set of denials which can be produced by using, e.g., De Morgan's laws).*

Example 4. Consider a relation $p(ISSN, TITLE)$ and let $U = \{p(\mathbf{i}, \mathbf{t})\}$ be the addition of a publication with title $\mathbf{t}$ and ISSN number $\mathbf{i}$ and $\phi = \leftarrow p(X,Y) \wedge p(X,Z) \wedge Y \neq Z$ the denial imposing uniqueness of ISSN. $\mathsf{After}^U(\{\phi\})$ is as follows:

$$\begin{aligned} &\{ \leftarrow [p(X,Y) \vee (X = \mathbf{i} \wedge Y = \mathbf{t})] \wedge [p(X,Z) \vee (X = \mathbf{i} \wedge Z = \mathbf{t})] \wedge Y \neq Z\} \\ \equiv &\{ \leftarrow p(X,Y) \wedge p(X,Z) \wedge Y \neq Z, \\ &\leftarrow p(X,Y) \wedge X = \mathbf{i} \wedge Z = \mathbf{t} \wedge Y \neq Z, \\ &\leftarrow X = \mathbf{i} \wedge Y = \mathbf{t} \wedge p(X,Z) \wedge Y \neq Z, \\ &\leftarrow X = \mathbf{i} \wedge Y = \mathbf{t} \wedge X = \mathbf{i} \wedge Z = \mathbf{t} \wedge Y \neq Z\}. \end{aligned}$$

Clearly, After's output is not in any "normalized" form, as it may contain redundant denials and sub-formulas (such as, e.g., $a = a$). Moreover, assuming that the original denials hold in the current database state can be used to achieve further simplification. For this purpose, a transformation $\mathsf{Optimize}_\Delta(\Gamma)$ is defined that exploits a given set of denials Δ consisting of trusted hypotheses to simplify the input set Γ. The proposed implementation [12] is described in [11] in terms of sound rewrite rules, whose application reduces denials in size and number and instantiates them as much as possible. For reasons of space, we refrain from giving a complete list of the rewrite rules in the Optimize operator and we describe its behavior as follows.

Given a set of denials Γ, a denial $\phi \in \Gamma$ is removed if it can be proved redundant from $\Gamma \setminus \{\phi\}$; ϕ is replaced by a denial ψ that can be proved from Γ if ψ subsumes ϕ; equalities involving variables are eliminated as needed. The resulting procedure is terminating, as it is based on resolution proofs restricted in size. The operators After and Optimize can be assembled to define a procedure for simplification of integrity constraints.

Definition 3. *For an update U and two sets of denials Γ and Δ, we define* $\mathsf{Simp}^U_\Delta(\Gamma) = \mathsf{Optimize}_{\Gamma\cup\Delta}(\mathsf{After}^U(\Gamma))$.

Theorem 1 ([11]). Simp *terminates on any input and, for any two set of denials Γ, Δ and update U,* $\mathsf{Simp}^U_\Delta(\Gamma)$ *holds in a database state D consistent with Δ iff Γ holds in D^U.*

We use $\mathsf{Simp}^U(\Gamma)$ as a shorthand for $\mathsf{Simp}^U_\Gamma(\Gamma)$.

Example 5. [4 cont.] The first denial in $\mathsf{After}^U(\{\phi\})$ is the same as ϕ and is thus redundant; the last one is a tautology; both the second and third reduce to the same denial; therefore the resulting simplification is $\mathsf{Simp}^U(\{\phi\}) = \{\leftarrow p(\mathbf{i}, Y) \wedge Y \neq \mathbf{t}\}$, which indicates that, upon insertion of a new publication, there must not already exist another publication with the same ISSN and a different title.

4.1 Examples

We now consider some examples based on the relational schema of documents `pub.xml` and `rev.xml` given in section 3.

Example 6. [1 continued] Let us consider constraint Γ from example 3 imposing the absence of conflict of interests in the submission review process. An update of interest is, e.g., the insertion of a new submission to the attention of a reviewer.

For instance, a submission with a single author complies with the pattern
$U = \{sub(\mathbf{i}_s, \mathbf{p}_s, \mathbf{i}_r, \mathbf{t}), auts(\mathbf{i}_a, \mathbf{p}_a, \mathbf{i}_s, \mathbf{n})\}$,

where the parameter ($\mathbf{i}_s$) is the same in both added tuples. The fact that $\mathbf{i}_s$ and $\mathbf{i}_a$ are new node identifiers can be expressed as a set of extra hypotheses to be exploited in the constraint simplification process:

$\Delta = \{ \leftarrow sub(\mathbf{i}_s, \_, \_, \_), \leftarrow auts(\_, \_, \mathbf{i}_s, \_), \leftarrow auts(\mathbf{i}_a, \_, \_, \_)\}$.
The simplified integrity check w.r.t. update U and constraint Γ is given by
$\mathsf{Simp}^U_\Delta(\Gamma)$: $\{\leftarrow rev(\mathbf{i}_r, \_, \_, \mathbf{n}), \leftarrow rev(\mathbf{i}_r, \_, \_, R) \wedge aut(\_, \_, I_p, \mathbf{n}) \wedge aut(\_, \_, I_p, R)\}$.

The first denial requires that the added author of the submission ($\mathbf{n}$) is not the same person as the assigned reviewer ($\mathbf{i}_r$). The second denial imposes that the assigned reviewer is not a coauthor of the added author $\mathbf{n}$. These conditions are clearly much cheaper to evaluate than the original constraints Γ, as they are instantiated to specific values and involve fewer relations.

Example 7. Consider the denial $\phi = \leftarrow rev(I_r, \_, \_, \_) \wedge \mathsf{Cnt}_{\mathsf{D}}(sub(\_, \_, \underline{I_r}, \_)) > 4$ imposing a maximum of 4 reviews per reviewer per track. The simplified integrity check of ϕ w.r.t. update U from example 6 is $\mathsf{Simp}^U_\Delta(\{\phi\}) = \{\leftarrow rev(\mathbf{i}_r, \_, \_, \_) \wedge \mathsf{Cnt}_{\mathsf{D}}(sub(\_, \_, \mathbf{i}_r, \_)) > 3\}$, which checks that the specific reviewer $\mathbf{i}_r$ is not already assigned 3 different reviews in that track.

5 Translation into XQuery

The simplified constraints obtained with the technique described in the previous section are useful only if they can be checked before the corresponding update, so as to prevent the execution of statements that would violate integrity. Under the hypothesis that the dataset is stored into an XML repository capable of executing XQuery statements, the simplified constraints need to be translated into suitable equivalent XQuery expressions in order to be checked. This section discusses the translation of Datalog denials into XQuery. We exemplify the translation process using the (non-simplified) set of constraints Γ defined in example 3. For brevity, we only show the translation of the second denial.

The first step is the expansion of the Datalog denial. It consists in replacing every constant in a database predicate (or variable already appearing elsewhere in database predicates) by a new variable and adding the equality between the new variable and the replaced item. This process is applied to all positions, but the first and the third one, which refer to element and parent identifiers and thus keeps information on the parent-child relationship of the XML nodes. In our case, the expansion is:

$$\begin{array}{l}\leftarrow rev(I_r, B, C, R) \wedge sub(I_s, D, I_r, E) \wedge auts(F, G, I_s, A) \\ \wedge aut(H, I, I_p, J) \wedge aut(K, L, I_p, M) \wedge J = R \wedge M = A\end{array}$$

The atoms in the denial must be sorted so that, if a variable referring to the parent of a node also occurs as the id of another node, then the occurrence as an id comes first. Here, no such rearrangement is needed. Then, for each atom $p(Id, Pos, Par, D_1, \ldots, D_n)$ where $D_1, \ldots, D_n$ are the values of tags d1, ..., dn, resp., we do as follows. If the definition of `$Par` has not yet been created, then we generate `$Id in //p` and `$Par in $Id/..`; otherwise we just generate `$Id in $Par/p`. This is followed by `$Pos in $Id/position()`, `$D1 in $Id/d1/text()`, ..., `$Dn in $Id/dn/text()`.

Then we build an XQuery boolean expression (returning `true` in case of violation) by prefixing the definitions with the `some` keyword and by suffixing them with the `satisfies` keyword followed by all the remaining conditions in the denial separated by `and`. This is a well-formed XQuery expression. Here we have:

```
some $Ir in //rev, $C in $Ir/.., $B in $Ir/position(),
     $R in $Ir/name/text(), $Is in $Ir/sub,
     $D in $Is/position(), $E in $Is/title/text(),
     $F in $Is/auts, $G in $F/position(),
     $A in $F/name/text(), $H in //aut, $Ip in $H/..,
     $I in $H/position(), $J in $H/name/text(),
     $K in $Ip/aut, $L in $K/position(),
     $M in $K/name/text()
satisfies $J=$R and $M=$A
```

Such expression can be optimized by eliminating definitions of variables which are never used, unless they refer to node identifiers. Such variables are to be retained because they express an existential condition on the element they are bound to. Variables referring to the position of an element are to be retained only if used in other parts of the denial. In the example, we can therefore eliminate the definitions of variables `$B`, `$C`, `$D`, `$E`, `$G`, `$I`, `$L`. If a variable is used only once outside its definition, its occurrence is replaced with its definition. Here, e.g., the definition of `$Is` is removed and `$Is` is replaced by `$Ir/sub` in the definition of `$F`, obtaining `$F in $Ir/sub/auts`.

Variables occurring in the `satisfies` part are replaced by their definition. Here we obtain the following query.

```
some      $Ir in //rev, $H in //aut
satisfies $H/name/text()=$Ir/name/text()
      and $H/../aut/name/text()=$Ir/sub/auts/name/text()
```

The translation of simplified version $\mathsf{Simp}^{U}_{\Delta}(\Gamma)$ is made along the same lines. Again, we only consider the simplified version of the second constraint (the denial $\leftarrow\ rev(\mathbf{i}_r,\_,\_,R) \wedge aut(\_,\_,I_p,\mathbf{n}) \wedge aut(\_,\_,I_p,R)$). Now, a parameter can occur in the first or third position of an atom. In such case, the parameter must be replaced by a suitable representation of the element it refers to. Here we obtain:

```
some      $D in //aut
satisfies $D/name/text()=%n
      and $D/../aut/name/text()= /review/track[%i]/rev[%j]/name/text()
```

where `/review/track[%i]/rev[%j]` conveniently represents $\mathbf{i}_r$. Similarly, `\%n` corresponds to $\mathbf{n}$. The placeholders `%i`, `%j` and `%n` will be known at update time and replaced in the query.

The general strategy described above needs to be modified in the presence of aggregates. Aggregates apply to sequences of nodes; therefore, the most suitable constructs to define such sequences are `let` clauses. In particular, there is a `let` clause for each aggregate. This does not affect generality, as variables bound in the `let` clauses correspond to the aggregate's target path expression possibly defined starting from variables already bound in the `for` clauses above. The expression is wrapped inside an `exists(...)` construct in order to obtain a boolean result; for this purpose, an empty `<idle/>` tag is returned if the condition is verified. Again, integrity is violated if the query returns `true`. Constraint $\leftarrow rev(I_r,\_,\_,\_) \wedge \mathsf{Cnt}_{\mathsf{D}}(sub(\_,\_,\underline{I_r},\_)) > 4$, shown in example 7, is mapped to XQuery as shown below.

```
exists( for $Ir in //rev  let $D := $R/sub  where count($D) > 4  return <idle/> )
```

The other constraints from the examples can be translated according to the same strategy.

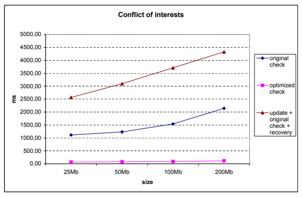
6 Evaluation

We now present some experiments conducted on a series of XML datasets matching the DTD presented in section 2, varying in size from 32 to 256 MB, on the examples described in order to evaluate the performance of our approach. Figures 1(a), 1(b) refer to the integrity constraints of examples 1, 2, respectively. The data were generated remapping data from the DBLP repository [10] into the schema of our running examples. Our tests were run on a machine with a 3.4 GHz processor, 1 GB of RAM and 140 GB of hard disk, using eXist [5] as XQuery engine. Execution times are indicated in milliseconds and represent the average of the measured times of 200 attempts for each experiment (plus 50 additional operations that were used as a “warm-up” procedure and thus not measured). The size of the documents is indicated in MB on the x-axis. Each figure corresponds to one of the running examples and reports three curves representing respectively the time needed to verify the original constraint (diamonds), verify the optimized constraint (squares) and execute an update, verify the original constraint and undo the update (triangles). We observe that we do not have to take into account the time spent to produce the optimized constraints, nor the cost of mapping schemata and constraints to the relational model, as these are generated at schema design time and thus do not interfere with run time performance. The curves with diamonds and squares are used to compare integrity checking in the non-simplified and, resp., simplified case, when the update is legal. The execution time needed to perform the update is not included, as this is identical (and unavoidable) in both the optimized and un-optimized case. The curve with triangles includes both the update execution time and the time needed to rollback the update, which is necessary when the update is illegal; when the update is illegal, we then compare the curve with triangles to the curve with squares. Rollbacks, needed since constraints are checked after an update, were simulated by performing a compensating action to re-construct the state prior to the update. The interpretation of these results is twofold, as we must consider two possible scenarios.

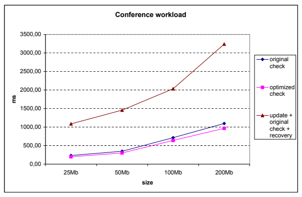
The update is legal: in the un-optimized framework the update is executed first and the full constraint is then checked against the updated database (showing that the update is legal); on the other hand, with our optimized strategy, the simplified constraint is checked first and the update is performed afterwards, since with our approach it is possible to check properties of the future database state in the present state, as discussed in section 4.

The update is illegal: in the un-optimized framework execution is as in the previous case, but this time the check shows that there is some inconsistency and, finally, a compensative action is performed. On the contrary, with our optimized strategy, the simplified constraint is checked first, which reports an integrity violation w.r.t. the proposed update; therefore the update statement is *not* executed.

From the experimental results shown in figures 1(a) and 1(b) it is possible to observe two features. The comparison between the performance of the optimized and un-optimized checks shows that the optimized version is always more efficient than



(a)



(b)

Fig. 1. Conflict of interests (a) and Conference workload (b)

the original one. In some cases, as shown in figure 1(a), the difference is remarkable, since the simplified version contains specific values coming from the concrete update statement which allow one to filter the values on which complex computations are applied. Further improvement is due to the elimination of a join condition in the optimized query. In other cases the improvement is not as evident because introduction of filters does not completely eliminate the complexity of evaluation of subsequent steps, such as the calculation of aggregate operations (figure 1(b)). The gain of early detection of inconsistency, which is a distinctive feature of our approach, is unquestionable in the case of illegal updates. This is prominently apparent in the cases considered in figures 1(a) and 1(b), since, as is well-known, the modification of XML documents is an expensive task.

7 Related work

Integrity checking is often regarded as an instance of materialized view maintenance: integrity constraints are defined as views that must always remain empty for the database to be consistent. The database literature is rich in methods that deal with relational view/integrity maintenance; Insightful discussions are in [7] and [4]. A large body of research in the field has also been produced by the logic programming and artificial intelligence communities, starting from [16]. Logic-based methods that produce *simplified* integrity tests can be classified according to different criteria, e.g., whether these tests can be checked before or only after the update, whether updates can be compound or only singleton, whether the tests are necessary and sufficient or only sufficient conditions for consistency, whether the language includes aggregates. Some of these methods are surveyed in [14]. In this respect, the choice of the simplification method of [11] seems ideal.

An attempt to adapt view maintenance techniques to the semi-structured data model has been made in [20] and in [17]. Incremental approaches with respect to validation of structural constraints have been proposed [1], as well as to key and foreign key constraints [3]. An attempt to simplification of general integrity constraints for XML has been made in [2], where, however, constraints are specified in a procedural fashion with an extension of XML Schema that includes loops with embedded `forbidden` assertions. We are not aware of other works that address the validation problem w.r.t. general constraints for XML. However, integrity constraint simplification can be reduced to query containment as long as the constraints can be viewed as queries. To this end, relevant works on containment are [18, 15]. There are several proposals and studies of constraint specification languages for XML by now. In [6], a unified constraint model (UCM) for XML is proposed

The XUpdate language that was used for the experimental evaluation is described in [8]. A discussion on update languages for XML is found in [21].

8 Conclusion and future works

In this paper we presented a technique for efficient constraint maintenance for XML datasets. We described the scenario in which integrity constraints are declaratively expressed in XPathLog, an intuitive logical language. These constraints are translated into Datalog denials that apply to an equivalent relational representation of the same data. Such denials are then simplified w.r.t. given update patterns so as to produce optimized consistency checks that are finally mapped into XQuery expressions that can be evaluated against the original XML document. Besides the possibility to declaratively specify constraints, the main benefits of our approach are as follows. Firstly, the ability to produce optimized constraints typically allows a much faster integrity checking. Secondly, performance is further improved by avoiding the execution of illegal updates completely: the optimized check is executed first and the update is performed only if the test guarantees that it will not violate integrity. In this paper we focused on updates whose contents are specified extensionally, as in the XUpdate language. More complex updates may be specified with a rule-based language such as XPathLog, i.e., intensionally in terms of other queries. Yet, introducing such updates would not increase complexity, as these are already dealt with by the relational simplification framework of section 4 and can be translated from XPathLog to Datalog as indicated in section 3.

Several future directions are possible to improve the proposed method. We are currently studying the feasibility of a trigger-based view/integrity maintenance approach for XML that would combine active behavior with constraint simplification. Further lines of investigation include integration of visual query specification tools to allow the intuitive specification of constraints; in this way domain experts lacking specific competencies in logic would be provided with the ability to design constraints that can be further processed with our approach.

Acknowledgements D. Martinenghi is partially supported by the TONES IST project FP6-7603 funded by the EU under the 6th Framework Programme.

References

1. A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.
2. M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated Update Management for XML Integrity Constraints. In *Inf. Proc. of PLAN-X Workshop*, 2002.
3. Y. Chen, S. B. Davidson, and Y. Zheng. Xkvalidator: a constraint validator for xml. In *CIKM ’02: Proc. of the 11th int. conf. on Information and knowledge management*, pages 446–452, New York, NY, USA, 2002. ACM Press.
4. G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
5. eXist. Open source native xml database. `http://exist.sourceforge.net`.
6. W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML. *Computer Networks*, 39(5):489–505, 2002.

7. A. Gupta and I. S. M. (eds.). *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
8. A. Laux and L. Matin. XUpdate working draft. Technical report, `http://www.xmldb.org/xupdate`, 2000.
9. A. Levy and Y. Sagiv. Constraints and redundancy in datalog. In *Proc. of the 11th PODS*, pages 67–80, New York, NY, USA, 1992. ACM Press.
10. M. Ley. Digital Bibliography & Library Project. `http://dblp.uni-trier.de/`.
11. D. Martinenghi. Simplification of integrity constraints with aggregates and arithmetic built-ins. In *Flexible Query-Answering Systems*, pages 348–361, 2004.
12. D. Martinenghi. A simplification procedure for integrity constraints. `http://www.ruc.dk/~dm/spic`, 2004.
13. W. May. XPath-Logic and XPathLog: a logic-programming-style XML data manipulation language. *TPLP*, 4(3):239–287, 2004.
14. E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *ER Workshops*, pages 62–73, 1999.
15. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *ICDT*, pages 315–329, 2003.
16. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
17. A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, 2005.
18. T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.
19. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, 1999.
20. D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *VLDB*, pages 227–238, 1996.
21. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

An Evaluation of the Use of XML for Representation, Querying, and Analysis of Molecular Interactions

Lena Strömbäck and David Hall

lestr@ida.liu.se
Department of Computer and Information Science,
Linköping University, Sweden

Abstract. Currently, biology researchers rapidly generate new information on how genes, proteins and other molecules interact in living organisms. To completely understand the machinery underlying life it is necessary to integrate and analyze these large quantities of data. As one step in this direction, new standards for describing molecular interactions have been defined based on XML. This work evaluates the usage of the XML Query language XQuery for molecular interactions, as it would be of great benefit to the user to work directly on data represented in the new standards. We use and compare a set of available XQuery implementations, eXist, X-Hive, Sedna and QizX/open for querying and analysis on data exported from available databases. Our conclusion is that XQuery can easily be used for the most common queries in this domain but is not feasible for more complex analyses. In particular, for queries containing path analysis the available XQuery implementations have poor performance and an extension of the GTL package clearly outperforms XQuery. The paper ends with a discussion regarding the usability of XQuery in this domain. In particular we point out the need for more efficient graph handling and that XQuery also requires the user to understand the exact XML format of each datase.

1 Introduction

During the past few years XML has become one of the most used formats for representation of information in a wide variety of domains and applications. In this paper we will discuss the current use of XML for molecular interactions, which is one important sub-area of bioinformatics. In this area the goal is to understand how proteins, genes, and other substances interact with each other within living cells. Proteins are the fundamental building blocks of life, and today biology researchers are gaining small pieces of information on each protein and how it interacts with other proteins and substances in the cell. To understand how the proteins and genes work together is the key to understanding the secret of life, and as such this has been set as a major goal for bioinformatics research by the Human Proteome Organization [8] and the US National Human Genome Research Institute [5], since this would be the key to new medical treatments for many diseases.

Within the area of molecular interactions the tradition has been to publish results from experiments on the web, making it possible for researchers to compare and reuse results from other research groups. This has resulted in a situation with a large number of available databases on Internet [2, 9, 12–15, 20, 26] with information about experimental results. However, the information content, data model and functionality is different between the different databases, which makes it hard for a researcher to track the specific information he needs.

There is, however, ongoing development within the field with the goal of making the datasets from each of the databases available for downloading and further analysis. Evaluations [1, 16] have shown that XML is beneficial for information representation within bioinformatics. Most of the existing molecular interaction databases allow export of data in XML. Recently, there have also been proposals for XML-based exchange formats for protein interactions, e.g. SBML [10], PSI MI [8], and BioPAX [3]. However, to allow for easy analysis and understanding of these datasets there is still a need for software for integration, querying and analysis based on XML.

The aim of this paper is to evaluate the use of available XML tools for direct usage of molecular interaction data available in XML. The paper starts with a brief introduction to the chosen data formats. After that we report on two experiments on analysis of data with XQuery. Finally we conclude the paper with a discussion on future needs for XML tools for this application.

2 XML standards for molecular interactions

There are currently a number of different XML formats for molecular interaction data available from different databases. In this work we will focus on the two formats SBML [10] and PSI MI [8]. We've chosen these formats because they have been proposed as future standards and there are currently large datasets of data available in these formats. Here, we give a short introduction to these standards; for a more extensive description and comparison with other formats see [23, 24].

Systems Biology Markup Language (SBML) [10] was created by the Systems Biology Workbench Development group in cooperation with representatives from many system and tool developers within the bioinformatics field. A brief example of an SBML model is given in Figure 1. As we can see, an SBML model contains a number of compartments, each of which is a description of the container or environment in which the reaction takes place. The substances or entities that take part in the reactions are represented as species. The interactions between molecules are represented as reactions, defined as processes that change one or more of the species. Reactants, products and modifiers for reactions are specified by references to the relevant species.

The Proteomics Standards Initiative Molecular Interaction XML format (PSI MI) [8] was developed by the Proteomics Standards Initiative, one initiative of the Human Proteome Organization (HUPO). An abbreviated example pathway represented in PSI MI is shown in Figure 1. In PSI MI the *experimentList* describes experiments and links to publications where the interactions are verified. The pathway itself is described via the *interactorList*, which is a list of proteins participating in the interaction, and the *interactionList*, a list of the actual interactions. For each *interaction* it is possible to set one or more names. The participating proteins are described by their names or by references to the *interactorList*. Note that, where the intention of SBML is to describe an actual interaction, i.e. that interacting substances produce some product, the purpose

SBML

```
<model name="Example">
 <listOfCompartments>
  <compartment name="Mithocondrial Matrix"
             id="MM">
 </listOfCompartments>
 <listOfSpecies>
  <species name="Succinate"
      compartment="MM" id="Succinate">
  <species name="Fumarate"
      compartment="MM" id="Fumarate">
  <species name="Succinate dehydrogenase"
      compartment="MM" id="Succdeh">
 </listOfSpecies>
 <listOfReactions>
  <reaction name="Succinate dehydrogenas
       catalysis" id="R1">
   <listOfReactants>
    <speciesReference species="Succinate">
   </listOfReactants>
   <listOfProducts>
    <speciesReference species="Fumarate">
   </listOfProducts>
   <listOfModifiers>
     <modifierSpeciesReference
                  species="Succdeh">
   </listOfModifiers>
  </reaction>
 </listOfReactions>
</model>
```

PSI MI

```
<entry>
 <interactorList>
  <proteinInteractor id="Succinate>
   <names>
    <shortLabel>Succinate</shortLabel>
    <fullName>Succinate</fullName>
   </names>
  </proteinInteractor> ...
 </interactorList>
 <interactionList>
  <interaction>
   <names>
    <shortLabel> Succinate dehydrogenas
         catalysis </shortLabel>
     <fullName>Interaction between ....
     </fullName>
    </names>
   <participantList>
    <proteinParticipant>
     <proteinInteractorRef ref="Succinate">
     <role>neutral</role>
    </proteinParticipant>
    <proteinParticipant>
     <proteinInteractorRef ref="Fumarate">
     <role>neutral</role>
    </proteinParticipant>
    <proteinParticipant>
     <proteinInteractorRef ref="Succdeh">
     <role>neutral</role>
    </proteinParticipant>
   </participantList>
  </interaction>
 </interactionList>
</entry>
```

Fig. 1. Examples of data in SBML and PSI MI

of PSI MI is to describe the result of an experiment, i.e. that there is some chemical interaction between the substances but roles of the substances in the interaction are not always known.

As we can see, the two formats are similar. Even so, there are several important differences between them. As previously discussed PSI MI contains more detailed information and there are differences in how participants in an interaction are represented. In addition to this there is also an important difference in the fact that SBML makes more use of XML attributes while PSI MI prefers to represent information as extra children in the tree structure. In the remainder of this paper we will look at possibilities for the researcher to work directly on the dataset, i.e. to analyze it by querying directly against the XML document.

1. *Find all information on a given compartment. Compartment id is given.*
2. *Find all information on a given species. Species id is given.*
3. *Find all information on a given reaction. Reaction id is given.*
4. *Count the number of species in the database.*
5. *Count the number of reactions in the database.*

Fig. 2. Queries for the SBML dataset

3 Experiment 1: XQuery querying

For our first experiment we want to test some common queries within the molecular interaction domain. For the experiments we use XQuery [32], the proposed standard query language for XML. The experiment consists of three parts: first the selection of queries and datasets for the test, next the formulation of the XQuery queries, and finally execution on XQuery implementations.

3.1 Definition of queries and datasets

Since the two standards SBML and PSI MI contain partly different information we define one set of interesting queries for each of the standards. All selected queries are based on an investigation of the query possibilities within available databases. The selected queries for SBML are presented in Figure 2. Since the PSI MI data model is richer than the one for SBML we could use more queries compared to the SBML dataset. Here we also wanted to test some combined queries, i.e. forcing XQuery to join information from different parts of the data file. The selected queries for the PSI MI dataset are presented in Figure 3.

1. *Find information on a given protein. Protein id is given.*
2. *Find information on a given experiment description. Experiment description id is given.*
3. *Find information on a given interaction. Interaction id is given.*
4. *Find the protein information for the proteins that participate in a given interaction. Interaction id is given.*
5. *Find the experiment description information for an experiment description that is part of an interaction. Interaction id is given.*
6. *Find all interactions that a given protein participates in. Protein id is given.*
7. *Find all interactions that a given experiment description is part of. Experiment description id is given.*
8. *Find any interactions that two given proteins are a part of. Protein ids for the two proteins are given.*
9. *Count the number of proteins in the database.*
10. *Count the number of interactions in the database.*

Fig. 3. Queries for the PSI MI dataset

The database currently providing the largest subsets of SBML data is Reactome [12]. It is a database on biological pathways, mainly human but there are pathways from other species as well. We selected Reactome's dataset for human reactions available in SBML level 2, version 1. The file is 3 MB in size. PSI MI is the most supported format for protein interaction databases. It is available as an alternative download format in a number of databases, for instance DIP [20], MINT [26] and IntAct [9]. Here, the IntAct database is the one currently providing the largest portions of PSI MI data. It is an open source database and toolkit for protein interactions. It currently contains nearly 40,000 interactions. We selected the dataset for Drosophila melanogaster for our tests. The dataset consists of nine files with the total size of 29.3 MB

3.2 Expressing queries in XQuery

In this section we discuss some issues in formulating queries for the first experiment. An extensive description of the queries is given in [7]. Many of the queries are written as simple path expressions with arbitrary depth (nesting) using // and some conditional. This is used in, for instance, the first three queries for both test cases which are very basic queries consisting of paths. Here we exemplify this with query 1 for PSI MI:

```
document("rat_small.xml")//proteinInteractor[@id="EBI-77471"]
```

For the more complicated queries, join over path expressions or in some cases the XQuery FLWOR expressions are used, since these make the queries more readable and easier to express. As an example of this we present query 4 for PSI MI which uses a FLWOR expression. Here we find the interaction with the appropriate ID and iterate over the proteins participating in this interaction. For each of these proteins we find the desired information in the list of proteininteractors.

```
for $ref in document("rat_small.xml")//interaction
  [names/shortLabel="interaction1"]
  /participantList/proteinParticipant/proteinInteractorRef/@ref
return document("rat_small.xml")//proteinInteractor[@id=$ref]
```

The last two queries for each standard use the count aggregate function to give a measure of the size of the datasets. Here we exemplify this with query 9 for PSI MI counting the number of proteins in the database.

```
count(document("rat_small.xml")//proteinInteractor)
```

From this discussion we can conclude that for a user that is accustomed to the concepts and constructions in the molecular interaction standards, and who has a reasonable knowledge of XQuery, these queries can easily be expressed.

3.3 Efficiency

Having formulated the queries, we were interested in the performance of the different XML database systems. There are a large number of implementations of XQuery, ranging from implementations for direct querying on XML files to systems aiming at more

Table 1. Response and load times for SBML.

Query	eXist	X-Hive	Sedna	QizX/open
	Mean time	Mean time	Mean time	Mean time
1	29 ms	6 ms	7 ms	838 ms
2	22 ms	2 ms	6 ms	875 ms
3	23 ms	8 ms	26 ms	859 ms
4	11 ms	71 ms	13 ms	724 ms
5	20 ms	72 ms	5 ms	719 ms
Load	9908 ms	2143 ms	1541 ms	-
Drop	1581 ms	4 ms	980 ms	-

Table 2. Response and load times for PSI MI.

Query	eXist	X-Hive	Sedna		Qizx/open
	Mean time	Mean time	Mean time	Worst case	Mean time
1	510 ms	2538 ms	68 ms	102 ms	6224 ms
2	329 ms	5 ms	5 ms	6 ms	6182 ms
3	172 ms	1291 ms	127 ms	589 ms	6339 ms
4	782 ms	1302 ms	182 ms	1119 ms	16 ms
5	602 ms	3153 ms	61 ms	63 ms	20115 ms
6	739 ms	223 ms	153 ms	1388 ms	6489 ms
7	2092 ms	60 ms	20 ms	148 ms	6459 ms
8	1308 ms	1758 ms	208 ms	927 ms	6443 ms
9	24 ms	2968 ms	12 ms	58 ms	4995 ms
10	73 ms	2945 ms	8 ms	9 ms	4995 ms
Load	167583 ms	23472 ms	23054 ms	-	-
Drop	14951 ms	176 ms	8829 ms	-	-

efficient storage and treatment of larger XML files, so-called native XML databases. We selected three native XML database implementations: eXist [27], Sedna [30] and X-Hive [31] and the XQuery API QizX/open [29] which does not support indexing and thus is expected to yield lower performance than the other systems.

Since the exact times can depend on external factors such as other processes running on the system the computer results would differ from time to time for the same queries. To decrease the influence of sudden spikes in measured time all queries were run several times and we base our values on the mean times. Table 1 presents the search times for SBML. The last two rows of the table show the times needed for loading the XML-file into the database and dropping it from the database.

For the SBML data set we can see that all the Native database systems have similar and good performance. QizX/open performs worse than the other systems for all queries, as expected.

For PSI the results are presented in Table 2. For this larger dataset the search times for Sedna were unequally distributed – the first search typically took somewhere between 5 and 30 times longer than the following searches. Due to this we also present the worst-case query for Sedna. Also, here we can see that the three Native XML systems perform well in principle with some surprising exceptions. For query 4 QizX/open, surprisingly, is the fastest system. Also, X-Hive has remarkably high query times for many of the queries – it does not seem to cope well with large amounts of data. The long query times for the count functions compared with those for Sedna and eXist may be an indication of different storage strategies for this type of data.

To conclude this section we can see that all the queries run with a reasonable response time on the selected datasets. Comparing the systems we can see that all the native databases provide similar performance with Sedna being the fastest and X-Hive being the most stable implementation.

4 Experiment 2: Pathway analyses

In addition to queries similar to those currently available through conventional systems, we also wanted to test advanced analysis on the datasets. In this case, we want to search for interaction chains between given proteins. As explained, it is not possible to make out reactant and products in PSI MI interactions since there is no order of the reactions defined in this format. Therefore we decided to concentrate the pathway searching efforts to the SBML dataset and used the following query:

Given two proteins find out if there is a given pathway between them in maximum n steps. Protein ids are given for the two proteins.

This analysis is very important, since it is often important to identify connections between interacting proteins in a given dataset. To express this in XQuery, recursion is required. The query is shown on next page. As we wanted to be able to test the query for various lengths of the pathways the query contains a recursive function *findMolecule* that returns elements for found connected proteins within a specified maximum length of the path. The function takes the start and goal reactants together with the cut-off depth as parameters.

```
declare function local:findMolecule($molecule as xs:string,
             $goalMol as xs:string, $n as xs:integer) {
  for $i in document("sbml.xml")//reaction
        [listOfReactants/speciesReference/@species=$molecule]
        /listOfProducts/speciesReference/@species
  return <item>{$i} {
    if($i = $goalMol) then <found/> else
      if($i = $molecule) then <loop /> else
        if ($n = 1) then <max/> else
          local:findMolecule($i, $goalMol, $n - 1)}
    </item>};
<path>{local:findMolecule("H2O", "sodium ion",2)}</path>
```

The response times for this query are presented in Table 3. eXist, Sedna and QizX/open ran out of memory at 4 steps. Sedna query times increase more slowly than for X-Hive, but X-Hive is the only XML tool reaching 4 steps.

These results are disappointing, both in the sense that formulating recursive XQuery queries gets rather complicated and that the response times are too high, especially taking into account that a real application would often need to do queries on larger datasets and longer paths than the ones we used.

For the molecular interaction applications there is a need for more efficient handling of these kinds of queries. One possibility would be to include an existing graph package into the XQuery language. For this reason we made an experiment with the graph package GTL (Graph Template Library) [28]. GTL is an extension of the Standard Template Library for graphs and graph algorithms in C++. The function we wanted to test, finding all paths between two proteins, was not implemented in GTL and we had to extend the package [7]. Before GTL can be used the protein interaction, data in SBML format is transformed to GML, a non-XML-based graph representation format provided by GTL. In our translation nodes are substances and edges are reactions. The transformation is done using the MSXSL command line transformation utility from Microsoft and it takes about 2 seconds to transform the 3 MB Reactome file.

The first step of our algorithm would be a search using GTL's built-in breadth-first search algorithm to verify that the end node really can be reached from the start node. The search between the start and end node is done by a recursive function, which works outwards from the start node and follows outgoing edges. In addition to this we use a cut-off depth at which to stop searching, as with the corresponding XQuery. The graph sent as an argument has already-visited nodes marked as hidden to avoid loops. Table 4 shows the performance of the pathway searches using our extension of GTL. The numbers given in the relevant table are based on a mean value of ten iterations.

As shown by this table the C++ program developed for graph searches is magnitudes faster than using the XQuery searches. This depends on a number of things with the most important probably being that the C++ program has graph and loop detection and that the representation is optimal for graph searches.

5 Discussion and implications for the future

The development of web databases and new standards within the area of molecular interaction indicates that XML representations will be of high importance for the area

Table 3. Test case 3: Pathway searches with XQuery.

Steps	eXist	X-Hive	Sedna	QizX/open
	Mean time	Mean time	Mean time	Mean time
1	422 ms	334 ms	121 ms	906 ms
2	951 ms	646 ms	245 ms	1125 ms
3	33323 ms	9053 ms	3493 ms	7172 ms
4	-	700443 ms	-	-

Table 4. Test case 3: Pathway searches using the extended GTL package

Steps	1	2	3	4	5	6	7	8
Time	0.3 ms	3 ms	15 ms	101 ms	823 ms	5,55 s	35,8 s	215 s

in the future. This means that there will also be a high degree of interest in existing XML technology as well as a need for development of new technology for the specific needs of the application. In this section we will put our results into context by providing a discussion on the generality of the results and requirements for the molecular interaction application.

For our experiments the most central criteria have been to test whether XQuery is useful for finding relevant information from a molecular interaction dataset with reasonable simple query formulations and a reasonable level of efficiency. To determine this we have based our queries on an investigation of available queries on existing databases for molecular interactions and cellular pathways available over the Internet [2, 9, 12–15, 20, 26]. This ensures that our selected queries capture the most important features of the application.

Another measure of generality is to compare our queries to available test benches for XML [4, 17, 21, 25]. These test benches define either particular XQuery queries or sets of XQuery queries, where the idea is to cover as many features of XQuery as possible. Such a comparison shows that our selected test queries cover the query groups relevant to this domain. Certain kinds of queries that were not relevant to this application or these datasets were naturally excluded from our tests, for example queries based on order.

For the queries in experiment 1, it was feasible to write queries on the protein-interaction data using the XQuery language. For a small dataset all three tested NXD's gave response times acceptable for interactive use with the exception for pathway queries. Our previous comparison with relational databases [22] also shows that these results are comparable with what can be achieved using a relational approach.

However, an interesting question is whether XQuery is a suitable query language for the domain. Even though it is possible to express queries, querying with XQuery requires a solid knowledge of the specific XML format of a dataset, which requires the user to have a high degree of knowledge about the specific XML formats. Since it is very likely that a typical user would need to handle several of these formats, there is a need for developing higher-level query languages for the domain.

In the case of more complex analyses, e.g. pathway analysis, the search times become large after just a few steps and the tested XQuery implementations do not cope with pathway queries longer than three steps. A C++ program for performing this query was developed using a graph package and resulted in searches that were orders of magnitude faster than for the XML databases, making it possible to search even larger graphs.

XQuery does not provide any special support for graph processing, while queries of this kind are doubtless very interesting in many applications. XGMML (Extensible Graph Markup and Modeling Language) [18] is a general format for describing graphs using XML based on GML, used in these tests. Other more specialized formats, such as the already-existing PSI MI and SBML formats for biological data, may emerge in a number of different subject areas.

This means there is a need for graph-capable XML in combination with XQuery. To be able to perform larger graph searches there must be special support for this. One possibility would be to extend XQuery with a number of internal functions for path searches and graph analyses. This is possible by using, for example, a Java binding such as those offered by eXist and QizX/open, which makes it possible to call functions in Java in the same way as XQuery's internal functions.

A final issue is the situation where the user needs to query over several datasets and integrate the resulting information into one query. Here we see several solutions. One is to provide a higher level query language capable of translating the query into several specific query languages. This is similar to what has been proposed for general databases within bioinformatics [11]. Another option would be to provide tools for fast data integration between the different XML formats in the line of the work within schema matching [6, 19].

6 Summary

XML is more and more commonly used within the area of molecular interactions and new XML standards are arising within the area. Because of this it would be very appealing if existing XML technology could be used for querying and analyses on this data. This work evaluates the use of XQuery and Native XML databases on datasets in two of the available standards, SBML and PSI MI.

Our experiments show that XQuery is a suitable language for most of the queries expected for the domain. We also saw a reasonable level of efficiency in the tested native XML implementations. There are, however, several obvious points for future research. One is the need for extended query languages and methods for graph analyses. A second issue is methods for the user to query over several different standard formats without having an exact knowledge of the specific XML format for each of the datasets.

Acknowledgements Thanks to Patrick Lambrix for valuable comments on this work.

References

1. Achard F, Vaysseix G, and Barillot E: XML, bioinformatics and data integration. *Bioinformatics* **17**(2):115-125, 2001.
2. Bader GD, Donaldson I, Wolting C, Oulette BF, Tony BF, Pawson TBF, and Hogue CWV: BIND - The Biomolecular Network Database. *Nucleic Acids Research* **29**(1):242-245, 2001.
3. BioPAX working Group: BioPAX – Biological Pathways Exchange Language. Level 1, Version 1.0 Documentation. Available at http://www.biopax.org, 2004.
4. Bressan S, Lee M-L, Li YG, Lacroix Z, Nambiar U: The XOO7 Benchmark. EEXTT 2002: 146-147, 2002.
5. Collins FS, Green ED, Guttmacher AE, and Guyer MS: A vision for the future of genomics research: A blueprint for the genomic era. *Nature* **422**: 835-847, 2003.

6. Doan, A, Halevy AY: Semantic Integration Research in the Database Community: A brief Survey. *AI Magazine, Special Issue on Semantic Integration, Spring 2005*, 2004.
7. Hall D. *An XML-based Database of Molecular Pathways.* Master Thesis, Department of Computer and Information Science, Linköpings universitet, LITH-IDA-EX–05/049–SE, 2005.
8. Hermjakob H, Montecchi-Palazzi L, Bader G, Wojcik J, Salwinski L, Ceol A, Moore S, Orchard S, Sarkans U, von Mering C, Roechert B, Poux S, Jung E, Mersch H, Kersey P, Lappe M, Li Y, Zeng R, Rana D, Nikolski M, Husi H, Brun C, Shanker K, Grant SGN, Sander C, Boork P, Zhu W, Akhilesh P, Brazma A, Jacq B, Vidal M, Sherman D, Legrain P, Cesareni G, Xenarios I, Eisenberg D, Steipe B, Hogue C, and Apweiler R: The HUPO PSI's Molecular Interaction format - a community standard for the representation of protein interaction data. *Nature Biotechnology* **22**(2):177-183, 2004.
9. Hermjakob H, Montecchi-Palazzi L, Lewington C, Mudali S, Kerrien S, Orchard S, Vingron M, Roechert B, Roepstorff P, Valencia A, Margalit H, Armstrong J, Bairoch A, Cesareni G, Sherman D, Apweiler R: IntAct - an open source molecular interaction database. *Nucl. Acids. Res.* 32: D452-D455, 2004.
10. Hucka M, Finney A, Sauro HM, Bolouri H, Doyle JC, Kitano H, Arkin AP, Bornstein BJ, Bray D, Cornish-Bowden A, Cuellar AA, Dronov S,. Gilles ED, Ginkel M, Gor V, Goryanin II, Hedley W-J, Hodgman TC, Hofmeyr J-H, Hunter PJ, Juty NS, Kasberger JL, Kremling A, Kummer U, Le Novere N, Loew LM, Lucio D, Mendes P, Minch E, Mjolsness ED, Nakayama Y, Nelson MR, Nielsen PF, Sakurada T, Schaff JC, Shapiro BE, Shimizu TS, Spence HD, Stelling J, Takahashi K, Tomita M, Wagner J, and Wang J: The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* **19**(4):524-531, 2003.
11. Jakoniene V, Lambrix P: Ontology-based Integration for Bioinformatics. Proceedings of the VLDB Workshop on Ontologies-based techniques for DataBases and Information Systems - ODBIS 2005, pp 55-58, Trondheim, Norway.
12. Joshi-Tope G, Gillespie M, Vastrik I, D'Eustachio P, Schmidt E, de Bono B, Jassal B, Gopinath GR, Wu GR, Matthews L, Lewis S, Birney E, Stein L: Reactome: a knowledgebase of biological pathways. *Nucleic Acids Res.* 1;33 Database Issue:D428-32. PMID: 15608231, 2005.
13. Kanehisa, M and Goto, S: KEGG: Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Res.* **28**, 27-30, 2000.
14. Kanehisa M, Goto S, Kawashima S, Okuno Y, and Hattori M: The KEGG resources for deciphering the genome. *Nucleic Acids Res.* **32**, D277-D280, 2004.
15. Karp P.D, Arnaud M, Collado-Vides J, Ingraham J, Paulsen IT, and Saier MH Jr: The E. coli EcoCyc Database: No Longer Just a Metabolic Pathway Database. *ASM News* **70**(1): 25-30, 2004.
16. McEntire R, Karp P, Abrenethy N, Benton D, Helt G, DeJongh M, Kent R, Kosky A, Lewis S, Hodnett D, Neumann E, Olken F, Pathak D, Tarzy-Hornoch P, Tolda L, and Topaloglou T: An evaluation of Ontology Exchange Languages for Bioinformatics. *Proceedings International Conference on Intelligent Systems for Molecular Biology* **8**:239-250, 2000.
17. Nambiar U, Lacroix Z, Bressan S, Lee M-L, Li YG: Current Approaches to XML Management. IEEE Internet Computing 6(4): 43-51, 2002.
18. Punin J, Krishnamoorthy M: XGMML (eXtensible Graph Markup and Modeling Language) http://www.cs.rpi.edu/∼puninj/XGMML/, 2001 (Accessed April 2005).
19. Rahm, E and PA Bernstein: A survey of approaches to semantic schema matching. *The VLDB Journal 10:334-350.* DOI 10-1007/s007780100057, 2001.
20. Salvinski L, Miller CS, Smith AJ, Bowie JU, and Eisenberg D: The Database of Interacting Proteins: 2004 Update. *Nucleic Acids Research* **32** Database Issue D449-451, 2004.

21. Schmidt AR, Waas F, Kersten ML, Florescu D, Carey MJ, Manolescu I, and Busse R: Why and How to Benchmark XML Databases. ACM SIGMOD Record, 3(30):27-32, 2001.
22. Strömbäck, L: Storage and Integration of Molecular Interactions: Evaluating the Use of XML Technology. *Proc of the 3rd International Workshop on Biological Data Management (BIDM'05)*. Copenhagen, Denmark, 2005.
23. Strömbäck, L: XML representations of pathway data: a comparison. *Proc. of the ACM SIGIR'04 Workshop on Search and Discovery within Bioinformatics.* Sheffield UK, 2004.
24. Strömbäck, L and Lambrix P: Representations of molecular pathways: An evaluation of SBML, PSI MI and BioPAX. Accepted for publication in *Bioinformatics* **21**(24):4401-4407, 2005.
25. Yao BB, Özsu MT, and Khandelwal N: XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of 20th International Conference on Data Engineering*, Boston, MA, pp 621-632, March 2004.
26. Zanzoni A, Montecchi-Palazzi L, Quondam M, Ausiello G, Helmer-Citterich M, and Cesareni G: MINT: a Molecular INTeraction database. *FEBS Letters* **513**(1):135-140, 2002.

27. eXist http://exist.sourceforge.net (Accessed May 2005)
28. GTL The Graph Template Library http://infosun.fmi.uni-passau.de/GTL/index.html (Accessed May 2005)
29. QizX/open http://www.xfra.net/qizxopen/ (Accessed May 2005)
30. Sedna http://www.modis.ispras.ru/Development/sedna.htm (Accessed May 2005)
31. X-Hivehttp://www.x-hive.com (Accessed May 2005)
32. XQuery http://www.w3c.org (Accessed May 2005)

Confidentiality Enforcement for XML Outsourced Data

Barbara Carminati and Elena Ferrari

University of Insubria at Como, Via Carloni, 78 - 22100 Como, Italy
email: barbara.carminati,elena.ferrari@uninsubria.it

Abstract. Data outsourcing is today receiving growing attention due to its benefits in terms of cost reduction and better services. According to such paradigm, the data owner is no more responsible for data management, rather it outsources its data to one or more service providers (referred to as publishers) that provide management services and query processing functionalities. Clearly, data outsourcing leads to challenging security issues in that, by outsourcing its data, the data owner may potentially loose control over them. Therefore, a lot of research is currently carriying on to ensure secure management of data even in the presence of an untrusted publisher. One of the key issues is confidentiality enforcement, that is, how to ensure that data are not read by unauthorized users. In this paper, we propose a solution for XML data, which exploits cryptographic techniques and it is robust to the most common and relevant security threats. In the paper, we present the encryption methods and query processing strategies.

1 Introduction

Data outsourcing is today emerging as one of the most important trend in the area of data management. According to such paradigm, the data owner is no more totally responsible for data management. Rather it outsources its data (or portions of them) to one or more publishers that provide data management services and query processing functionalities. Main benefit of data outsourcing is cost reduction for the owner, in that it pays only for the services it uses and not for the deployment, installation, maintenance, and upgrades of DBMSs. By contrast, when data are outsourced such costs are amortized across several users. Another important benefit is scalability in that data owners could not become bottlenecks for the system, rather they can outsource their data to as many publishers as they needs according to the amount of data and the number of managed users. Clearly, data outsourcing leads to many research challenges. One of the most significant is related to security. The key problem is that, since the owner does not anymore manage its data, it may potentially loose control over them. The challenge is therefore how to ensure the most important security properties (e.g., confidentiality, integrity, authenticity) even if data are managed by a third party. A naive solution is to assume the publisher to be *trusted*, that is, to assume it always operates according to the owner's security policies. However, making this assumption is not realistic, especially for web-based systems that can be easily attacked and penetrated. Additionally, verifying that a publisher is trusted is a very costly operation. Therefore, the research is now focusing on techniques to satisfy main security properties even in the presence of an untrusted publisher that can not always follow owner's security policies (for instance it can maliciously modify/delete the data it manages or it can send data to non authorized users). In this paper, we make a contribution to this research by focusing on confidentiality, that is, protection against non authorized reading operations, since it represents one of the most relevant security properties. Moreover, we cast our techniques in the framework of XML [11], since it is today the de-facto standard for data modeling and exchange over the web.

When data are outsourced confidentiality has two main aspects. The first, which we call *confidentiality wrt users*, refers to protect data against unauthorized read operations by users. The second, which we call *confidentiality wrt publishers*, deals with protecting owner's data from read operations by publishers. Our solution enforces confidentiality by using cryptographic techniques: publishers manage ciphered data instead of clear-text ones. In that way confidentiality wrt publishers is always ensured. Data encryption is generated by the owner and it is driven by the specified access control policies: all data portions to which the same policies apply are encrypted with the same key. Then, each user receives only the keys corresponding to the policies he/she satisfies. This ensures confidentiality wrt users. Enforcing confidentiality through the use of encryption requires dealing with several issues. The first is encryption generation in that there is the need to devise an encryption scheme which is robust to the most relevant security attacks. For instance, if you consider keyword-based searches on textual data, if the same keyword is always encrypted with the same key, both publishers and users can infer information by analyzing the document encryption. The same problem arises at the schema level, that is, with encrypted tags or attributes names. In this work we propose an encryption scheme that avoids information leakage due to data dictionary attacks.

The second main issue is how publishers can query encrypted data. In the literature, several methods exist to this purpose.[1] Some of them have been designed for relational databases [6, 7], others have been designed to query textual data [10]. Our work is inspired by both of them, since an XML document contains both text and attributes with standard domains. In the paper, we present our strategy to query XML encrypted data and we describe both publisher side and user side query processing.

The work described in this paper is part of a wider project whose goal is to ensure the most relevant security properties (i.e., authenticity/integrity and completeness in addition to confidentiality) when data are managed by a third party. Therefore, before going into the details of confidentiality protection we describe the overall architecture of the system we propose. Details on techniques for authenticity/integrity and completeness verification can be found in [1].

System architecture is depicted in Figure 1. Some of its components will be explained later on in the paper. In our system, users submit queries through a *client*, i.e., a program that users download from the owner site, and which makes them able to submit encrypted queries to publishers, and verify security properties on the received answers.

Security properties verification requires that some additional information is transmitted by the owner to both publishers and users. To limit the overhead that this information transfer requires, such information (which is coded in XML) is stored by the owner into a directory server (which can also be shared among different owners belonging to the same domain or to federated ones). The directory server contains three

[1] Some of them are surveyed in Section 2.

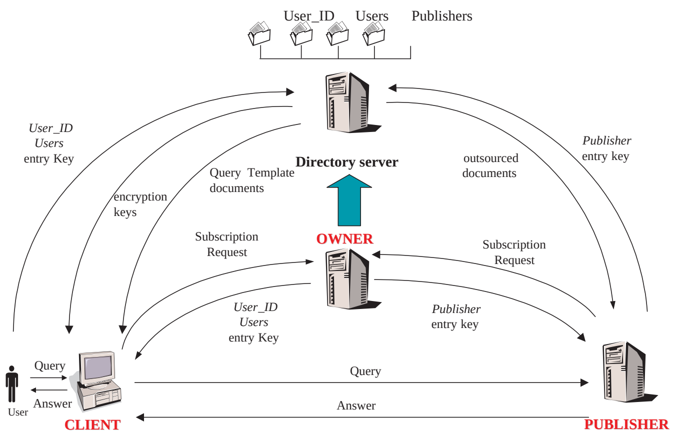


Fig. 1. Overall architecture [4]

kinds of entries: the *publishers* entry, that is shared by all publishers and contains all the ciphered documents that they are entitled to manage, plus additional information they need for the correct functioning of the system; the *users* entry, shared by all the subscribed users, storing common information; and a distinct *user_id* entry for each single user, containing needed information to verify security properties on the received answers. Both publishers and users are required to register once to the owner. After subscription, they receive the keys to access corresponding entries in the directory.

The remainder of this paper is organized as follows. Next section surveys related work that are the basis of our proposal. Section 3 deals with confidentiality enforcement. Section 4 illustrates client-side query processing. Finally, Section 5 concludes the paper.

2 Related work

In recent years the problem of inquiring encrypted data managed by a third party has been deeply investigated by several researchers, that proposed different solutions [6, 7, 10] for different data models and domains.

In general, the most appropriate solution to query encrypted data mainly depends on the nature of the data being queried, that is, the data domain and the underlying data model. Thus, in order to select the most appropriate technique to inquire XML encrypted data, we need to take into account the characteristics of XML data. XML documents often contain data with heterogeneous domains (e.g, textual data, date, integer). For this reason, we believe that in the scenario considered in this paper, it is not enough to use a single technique to inquiry encrypted XML data. Rather, different techniques should be combined into a unified framework to manage data with different domains. For instance, the work by Hacigumus et al. [6, 7] develops a method to query encrypted data stored in relational databases. From such work we borrow the method to query non textual data. By contrast [10] deals with keyword-based searches on textual data. We use some of the methods proposed in [10] to solve a twofold issue, that is, querying textual data and avoiding information leakage due to data dictionary attacks at the schema level. However, with difference to our proposal, such approaches only consider confidentiality wrt publishers, whereas they do not consider confidentiality wrt users. In what follows, we briefly introduce the techniques proposed in [6, 7] and [10], whereas details on how these two techniques are used in our framework are presented in Section 3.

Confidentiality enforcement through the use of encryption techniques has also been investigated by us in [4]. The current work extends our previous work along several relevant directions. The first is that in this paper we provide a comprehensive method to query encrypted XML documents, which exploits a variety of strategies to query XML documents with heterogeneous content. In [4] we adapt the strategy proposed in [6, 7] to query both textual and non textual attributes as well as data with textual domain. The technique proposed in [6, 7] has a major shortcoming, i.e., to be prone to data dictionary attacks by both publishers and users. This kind of attack can be for instance perpetrated when the same data portion (e.g., attribute value, tag name) repeatedly appears in an XML document ciphered with the same key. In this paper, we solve this problem by applying a combination of strategies to treat XML data, able to trade-off between query expressivity and robustness to security threats. Another weak point of the solution proposed in [4] is information leakage due to inferences at the schema level in that tags/attributes with the same names and covered by the same access control policies are encrypted with the same key. Therefore, by analyzing document encryptions both publishers and users can infer information on the schema of some document portions, even if they are not allowed to access such portions. This is a relevant security threat since tags and attributes names can convey semantic relevant information (for instance, by exploiting such kind of attack a user can infer the existence of an element named `salary` even if he/she is not allowed to access it according to owner's security policies).

2.1 Hacigums et al.

The approach proposed by Hacigumus et. al [6, 7] exploits binning techniques and privacy homomorphisms to inquiry encrypted relational data. Binning techniques are used to perform selection queries on encrypted relation data, whereas privacy homomorphisms are used to make a third party able to perform aggregate queries over encrypted tuples. In our framework, we assume that users submit queries through XPath [11], with the obvious intention to extend the approach to support XQuery [11]. Thus, since XPath expressions do not contain aggregate functions, in what follows we review only the approach for selection queries.

The underlying idea of the approach is the following: given a relation *R*, the owner divides the domain of each attribute in *R* into distinguished partitions, to which it assigns a different id. Then, the owner sends the publisher the encrypted tuples, together with the ids of the partitions corresponding to each attribute value in *R*. The publisher is able to perform queries directly on the encrypted tuples, by exploiting the received partition ids. The idea is that a user, before submitting a query to a publisher, rewrites it in terms of partition ids. As an example, consider the relation $Employee(eid, ename, salary)$, and, for simplicity, consider only the $salary$ attribute. Suppose that the domain of $salary$ is in the interval [500k, 5000k], and that an equi-partition with 100k as range is applied on that domain. Suppose that a user wants to perform the following query: "SELECT * FROM Employee WHERE salary =1000k". It translates it into the query: "SELECT * FROM Employee WHERE salary=id_{1000k}", where id_{1000k} is the id of the partition containing the value 1000k. Clearly, users should receive by the owner information on the techniques used to partition data and generate ids. The publisher is then able to answer such query by exploiting only the received ids. The publisher returns an approximate result wrt the original query. For instance, with reference to the above example, it returns all the tuples of the *Employee* relation whose *salary* attribute belongs to the range [1000K, 1099K). A further query processing has thus to be performed by the user to refine the received answer.

2.2 Song et al.

Another approach that has greatly inspired the present work is the one proposed by Song et al. [10] for keyword searching on encrypted textual data. Given a text consisting of a sequence of words: $W_1, W_2, \ldots W_n$, the basic scheme proposed in [10] first encrypts each word using a symmetric encryption algorithm $E_k()$, with a single secret key k. Then the scheme generates the XOR of each encrypted word with a pseudorandom number. The resulting ciphered words are then outsourced to the third party. According to this scheme, when a user needs to search for a keyword W, it generates the encrypted word $E_k(W)$ and computes $E_k(W) \oplus S$, where S is the corresponding pseudorandom number. This simple scheme allows the third party to search for keyword W in the encrypted data, by simply looking for $E(W) \oplus S$, thus without gaining any information on the clear text. Since occurrences of the same word are xored with different pseudorandom numbers, by analyzing the distribution of the encrypted words, no information could be inferred regarding the clear text.

In particular, in [10] the authors propose a generation process for pseudorandom numbers, that makes users able to locally compute pseudorandom numbers, without any interaction with the data owner. The scheme exploits a symmetric encryption function $E()$, and two pseudorandom numbers generator functions, namely F and f. In the following, with the notation $E_k(x)$ ($F_k(x)$, $f_k(x)$, respectively), we denote the result of applying E (F, f, respectively) to input x with key k. The scheme considers as input a set of clear-text words, $W_1, W_2, \ldots, W_l$, with the same length n.[2] Given these set of words, the steps needed to generate the corresponding ciphered words are the following:

- data owner generates a sequence of pseudorandom values: $S_1 \ldots S_l$, of length $n - m$;[3]
- for each word W_j, the outsourced ciphered word C_j is generated according to the following formula: $C_j = E_k(W_j) \oplus < S_j, F_{K_j}(S_j) >$, where $K_j = f_k(FB_j)$, and FB_j denotes the first $n - m$ bits of $E_k(W_j)$.

[2] This set of words can be obtained by partitioning the input clear-text into atomic quantities (on the basis of the application domain), and by padding and splitting the shortest and longest words.

[3] Parameter m can be properly adjusted to minimize the number of erroneous answers due to collision of pseudorandom numbers generator $F()$ and $f()$.

Let us see now how query evaluation and decryption take place. When a user needs to search for a keyword W_i, he/she sends the third party $E_k(W_i)$ and key K_i, which can be locally computed by the user. Then, for each outsourced ciphered word C_i, the third party: *1)* calculates $C_i \oplus E_k(W_i)$; *2)* takes the first $n-m$ bits bts of the resulting value, and computes $F_{K_i}(bts)$, where K_i is the key received by the requiring user; *3)* if the result of $F_{K_i}(bts)$ is equal to the $n-m+1$ remaining bits, then the ciphered word C_i is returned. Indeed, if C_i contains the searched encrypted word, then $E_k(W_i) \oplus < S_i, F_{K_i}(S_i) > \oplus E_k(W_i) = < S_i, F_{K_i}(S_i) >$, for the properties of the XOR operator. When a user receives C_i as answer of a query, he/she is not able to extract the value $E_k(W_i)$ from C_i and thus to decrypt it, by simply using the decryption key k. Therefore, the scheme proposed in [10] assumes that users know S_i.[4] In such a way, user is able to recover FB_i, by xoring S_i with the first $n-m$ bits of C_i. Having FB_i, the user can generate K_i (i.e., $K_i = f_k(FB_i)$)), which can be used to compute $F_{K_i}(S_i)$. Finally, having $< S_i, F_{K_i}(S_i) >$ the user is able to extract from C_i the encrypted word $E_k(W_i)$, and to decrypt it.

3 Confidentiality enforcement

Enforcing confidentiality in an outsourced-based architecture requires to address confidentiality wrt both publishers and users. Since publishers could be untrusted, it is necessary to devise some mechanisms to avoid their malicious usage of owner's data. On the other hand, users are usually entitled to access only selected portions of owner's data based on their characteristics and profiles.

Confidentiality requirements wrt users are usually modelled through a set of access control policies stating who can access what portions of the owner's data. In traditional client-server architectures confidentiality wrt users is usually enforced by means of access control mechanisms (i.e., reference monitors), which mediate each user access request by authorizing only those in accordance with the owner's access control policies. A fundamental requirement is therefore the presence of a trusted environment hosting the reference monitor. In traditional scenarios, this environment is provided by the entity managing the data, i.e., the owner's DBMS server. Enforcing access control in a third party scenario would imply the delegation of the reference monitor tasks to publishers. However, since we are considering a scenario where assumptions on publisher trustworthiness cannot be done, such solution is no longer applicable.

To enforce confidentiality wrt both users and publishers we therefore propose an alternative solution based on cryptographic techniques. The underlying idea is that data owners outsource to publishers an encrypted version of the data they are entitled to manage, without providing them the corresponding decryption keys. Such encryption is generated in such a way to minimize the risks of data dictionary attacks. Therefore, the publisher is not able to access and, as a consequence, to misuse the outsourced data, since they are ciphered. To enforce confidentiality wrt users we propose a particular document encryption, hereafter called *well-formed encryption*, where different portions of the same document are encrypted with different keys, on the basis of the access control policies specified by the owner. In order to correctly enforce access control, it is

[4] Users are able to generate it using the pseudorandom number generator and knowing the seed.

therefore necessary to selectively distribute secret keys to users. According to the architecture presented in Figure 1, the appropriate keys are stored in the owner directory server, in such a way that each user obtains all and only the keys corresponding to the policies he/she satisfies. Policies are specified according to a credential-based access control model for XML data proposed by us in [2]. Appropriate keys for each user are therefore determined by evaluating user credentials against the specified policies. The well-formed encryption and the selective distribution of secret keys ensure confidentiality wrt users. Indeed, each node of the resulting encrypted document is accessible only to authorized users, that is, those users who have received the appropriate keys directly by owners. Even in the case that an untrusted publisher returns unauthorized nodes to a user, he/she is not able to access it, since he/she has not been provided with the corresponding keys.

Applying a cryptographic-based solution in a third party scenario implies to address two main issues. The first is related to the fact that publishers operate on ciphered data. Therefore, we need some mechanisms to make them able to perform queries over them. In Section 3.1, we show how Hacigumus et al. and Song et al. approach can be adopted in the context of XML. The second issue is the definition of encryption strategies able to reduce as much as possible security threats that can be perpetrated against the system. In particular, we have addressed information inference threat that a publisher (user) can perpetrate by analyzing the distribution of encrypted nodes. To overcome this drawback, in Section 3.2, we show how Song et al. scheme can be adopted to encrypt tagnames and attribute names and values, thus to avoid inference.

3.1 Inquiring encrypted XML data

As introduced in Section 2, selecting the most appropriate solution to query encrypted data mainly depends on the nature of the data, that is, the data domain and the underlying data model. Usually, an XML document contains data to be modelled into elements, which in turn could contain other elements in accordance to the structure of the modelled data. Whereas attributes are usually exploited to better describe data contained into the corresponding elements. Let us consider, for instance, an XML document modelling a book. It is reasonable to suppose that in such an XML document, `Chapter` elements contain book's chapters, whereas their attributes `Title` and `Number` store additional information about book's chapters. Therefore, in devising methods to query XML encrypted data, we need to consider that elements could be searched based on their attributes values and/or their contents. Thus, we need a method that makes the publisher able to perform logical comparisons on encrypted attribute values, as well as keyword-based searches on encrypted element contents and textual attributes.

To cope with this requirement, we have adopted two different strategies to query XML encrypted data. In particular, we use an approach similar to the one proposed by Song et al.'s [10] to query elements and attributes with textual domain. By contrast, for non textual elements and attributes, we exploit the method proposed by Hacigumus et al. [6]. Thus, we assume that before encrypting a node *n*, the encryption process determines *n*'s data domain. This task can be performed by means of information contained in the XMLSchema. Let us see in more details which are the query strategies for textual and non-textual encrypted data.

Textual data. Song et al.'s approach [10]. In particular makes a publisher able to search for a specific keyword on encrypted textual data without loss of data confidentiality. Applying such an approach to our scenario requires two adjustments wrt the original formulation. The first is because the scheme proposed in [10] works for words of the same length, whereas we need to consider words of variable length. Therefore, we adapt the scheme proposed in [10] to the management of variable length words, as follows. Let **W** be the longest word in the owner's dictionary, and let $L_{\mathbf{W}}$ be the length of **W**. Thus, for each sequence of clear-text words: $W_1, W_2, \ldots, W_l$, with length $L_j \leq L_{\mathbf{W}}$, the steps for ciphered words generation are the following:

- for each word W_j, data owner pads it with a sequence $pdbts$ of $L_{\mathbf{W}} - L_j$ bits;
- data owner generates a sequence of pseudorandom values $S_1 \ldots S_l$, of length $L_{\mathbf{W}} - m$;
- for each keyword W_j the outsourced ciphered word C_j is generated by the following formula: $C_j = E_k(W_j || pdbts) \oplus < S_j, F_{K_j}(S_j) >$, where $K_j = f_k(FB_j))$, and FB_j denotes the first $n - m$ bits of $E_k(W_j || pdbts)$.

According to this scheme, users have to know $L_{\mathbf{W}}$, that is, the length of the longest word managed by the data owner, to be able to pad the searched keyword W_i, before submitting the query.

Finally, we need to define a method for keywords selection. In particular, given a node n, we need to state how to select keywords from n's content. A naive solution is to split the content into separate $L_{\mathbf{W}}$ blocks, and to treat them as distinguished keywords. This solution however is useless if we consider that the resulting ciphered words should be exploited for the search. Thus, it is obvious that some content analysis should be performed over the node's content before keywords selection. In particular, the solution we propose requires a first phase during which the owner preprocesses the textual data contained into an XML node and extracts a set of keywords.[5] Then, each keyword is ciphered according to the scheme introduced above.

Non-textual data. To make publishers able to evaluate queries on encrypted non-textual data, we adapt Hacigums et al.'s approach [6]. First, we have to deal with partition generation. In general, the choice of the most appropriate partitioning technique mainly depends on the node domain. For instance, for numeric data (such as integer, real, etc.), a strategy based on an equi-partitioning of the domain could be appropriate. By contrast, for temporal data a partitioning based on time intervals could be more appropriate. Therefore, in our system we associate a different partitioning function with each possible data domain, with the exception of textual domain. Thus, given a node n of a document d, by analyzing the XML schema defining d it is possible to select the appropriate partitioning function. The partitioning function takes as input the node value and returns the id of the partition to which the node belong to, generated according to the data domain.

[5] Several techniques developed in the Information Retrieval field can be used to this purpose [9].

```
<Sec-Info>
  <Node-Info  Name='$C_{K_1}(CD)$'>
    < Query-Info> ...</Query-Info>
  </ Node-Info>
  <Attributes>
    <Node-Info  Name='$C_{K_2}(Price)$'  Value='$C_{K_2}(30)$'>
      <Query-Info> ... </Query-Info>
    </ Node-Info>
  </Attributes>
</Sec-Info>
```

Fig. 2. An example of `Sec-Info` element

3.2 Document encryption

Document encryption requires a first phase during which each node of the input document is associated with the proper secret key. Therefore, each node of the input document is first marked with the set of access control policies applied to it, and then a different secret key is generated for each different configuration of policies that applies to a document portion. Then, all the nodes are encrypted with the corresponding secret keys.

In particular, we use an encryption strategy that preserves as much as possible the original structure of the XML document. In the well-formed encryption, the encryption of an XML element e is an XML element $\overline{e}$ having as tagname the encryption of $e.tagname$[6] and as element content the encryption of $e.content$. The well-formed encryption is therefore an XML document that preserves the elements relationships of the original document. We have decided to preserve as much as possible the structure of the original document in the well-formed encryption since this simplifies query formulation. Indeed, users formulate queries through XPath expressions that exploit the structure of the XML document.

However, preserving the original document structure implies some security threats. An untrusted publisher (or user) could infer information by analyzing the distribution of encrypted nodes. This threat could be easily perpetrated against tagnames, attribute names and values, since an XML document may often contain repeated elements and several attributes with the same names (or values). To overcome this drawback, rather than symmetric encryption we apply the scheme proposed by Song et al. (see Section 2.2) to encrypt tagnames and attribute names and values. By contrast, for element contents we have decided to adopt traditional symmetric encryption (e.g., TDES, AES), that requires less computational resources. This choice is motivated by the fact that probability of having a number of occurrences of the same encrypted element content sufficient for data dictionary attacks is small. This probability is further reduced in our context, where elements with the same content are encrypted with different keys, if they are protected by different access control policies.

[6] In the paper, given an element e (i.e., an attribute a) we use the dot notation to identify its tagname (name) $e.tagname$ (i.e., $a.name$) and content (value) $e.content$ (i.e., $a.value$).

Algorithm 1 *The element encryption algorithm*

```
INPUT:
    1. An XML element $e$
    2. The encryption key $K$ corresponding to the policy configuration applied to $e$
OUTPUT:
    An XML element, $\bar{e}$, containing the encryption of $e$ and its query processing information

1. Let $\bar{e}$ be an empty XML node;
2. Set $\bar{e}.tagname$ equal to $C_K(e.tagname)$;
3. Set $\bar{e}.content$ equal to $E_K(e.content)$;
4. Let $se$ be an empty Sec-Info element;
5. Let $ni$ be an empty Node-Info element subelement of $se$;
6. Set the Name attribute of $ni$ equal to $C_K(e.tagname)$;
7. If the domain of $e$ is textual:
       Let $WS$ be the set of keywords extracted from $e.content$;
       For each $w \in WS$: Insert $C_K(w)$ into the Query-Info subelement of $ni$;
   Else
       Let $PF()$ be the partitioning function associated with $e$'s domain;
       Insert $PF(e.content)$ into the Query-Info subelement of $ni$;
8. Return $\bar{e}$;
```

To make publishers able to evaluate queries on encrypted XML documents, the resulting document encryption is complemented with additional information, called *query processing information*. This information consists of partition's ids or ciphered keywords associated with attribute values or element contents. Query processing information are encoded by an XML element, called `Sec-Info`, which is inserted into each element of the well-formed encryption (see Figure 2) and encodes query processing information of both the element itself and all its attributes. The `Sec-Info` element associated with an element *e* contains a mandatory subelement, named `Node-Info`, whose `Query-Info` subelement stores the query processing information corresponding to *e*. The `Query-Info` subelement contains the ciphered words extracted from *e.content* or the id of the partition to which *e* belongs to, if element *e* has non-textual domain. By contrast, to model query processing information corresponding to each attribute of element *e*, the `Sec-Info` contains an `Attributes` subelement, which in turn contains a different `Node-Info` subelement for each attribute of *e*. The `Node-Info` subelement corresponding to attribute *a* of element *e* contains a `Query-Info` subelement storing the id of the partition to which the value of *a* belongs to or the ciphered keywords extracted from it, if attribute *a* has textual domain.

Let us now see in more details how given a document *d*, the generation of $\overline{d}$, that is, the document to be outsourced, is carried on. This process is realized by means of two different phases. During the first phase, each element *e* of *d* is encrypted, according to the strategy previously explained, and inserted into $\overline{d}$. Additionally, the `Sec-Info` element associated with *e* is generated. Then, attribute encryption is performed and the `Sec-Info` element is updated accordingly. Algorithm 1 deals with element encryption. The algorithm takes as input an element *e* and the encryption key corresponding to the policy configuration applied to *e*. First, it generates the encryption of the tagname and element content (steps 2 and 3). This is done by means of $C_K()$ function implementing the Song et al.'s scheme, and $E_K()$ function that performs symmetric encryption. Then, Algorithm 1 generates the query processing information associated with the input element. In particular, in case of textual domain, the system extracts the set of meaningful keywords from $e.content$ and generates the corresponding ciphered keywords, which are then inserted into the `Query-Info` element. By contrast, for element with non-textual domain, the query processing information consists of the id of the partition to which e belongs to.[7]

In order to complete the well-formed encryption we need to consider all attributes of e. This is done during a second phase, implemented by Algorithm 2. This phase considers each attribute a and generates the encryption of its name and value, according to Song et al.'s scheme. Moreover, it generates the query processing information related to a, in the same way as Algorithm 1. The resulting information is then inserted into the `Sec-Info` element associated with the element to which a belongs to. This information is stored as a new `Node-Info` element inside the `Attributes` subelement of `Sec-Info`.

Algorithm 2 *The attribute encryption algorithm*

```
INPUT:
  1. An XML attribute $a$
  2. The encryption key $K$ corresponding to the policy configuration applied to $a$
OUTPUT:
  The updated Sec-Info containing the encryption of $a$ and its query processing information

1. Let $f$ be the element containing $a$;
2. Let $se$ be the Sec-Info element in $f$;
3. Let $ni$ be an empty Node-Info element;
4. Set the Name attribute of $ni$ equal to $C_K(a.name)$;
5. Set the Value attribute of $ni$ equal to $C_K(a.value)$;
6. If $a$ has a textual domain:
      Let $WS$ be the set of keywords extracted from $a.value$;
      For each $w \in WS$: Insert $C_K(w)$ into the Query-Info subelement of $ni$;
   Else
      Let $PF()$ be the partitioning function associated with $a$'s domain;
      Insert $PF(a.value)$ into the Query-Info subelement of $ni$;
7. Insert $ni$ in the Attributes subelement of $se$;
8. Return $se$;
```

4 Client side query processing

In this section we show how the client is able to submit queries to publisher and decrypt the resulting nodes. Before going into the details we introduce the *query template*.

[7] We assume that our system manages a library of partitioning functions, which associates with each different data domain a unique partitioning function.

4.1 Query template

The query template has a twofold goal. The first is to make a user able to verify the completeness of a query result,[8] the second is to make a user able to formulate queries and to decrypt the received results. We do not go into the details of completeness verification, since it is outside the scope of this paper (interested readers could refer to [4]). Rather, we focus on the information the query template contains for query processing.

Query templates are generated by the owner for each outsourced document and are stored by the owner in the users' directory entry, and thus available to all users for downloading. Query template contains the encrypted structure of the corresponding XML document and it is generated by using the same encryption strategy employed for XML documents. Moreover, the query template contains the encrypted pseudorandom numbers associated with each node, and needed by clients for document decryption. More precisely, all additional information associated with an element *e* needed for both query processing and completeness verification are encoded by an XML element, which is inserted as direct child of *e* into the query template. The element is defined according the syntax of the `Sec-Info` element (cfr. Section 3.2). All information is therefore stored into the `Node-Info` element. Each `Node-Info` element contains an additional attribute called `N` storing the encrypted pseudorandom number associated with the element tagname or attribute name to which the `Node-Info` element refers to.[9] Pseudorandom number is encrypted with the encryption key associated with the element/attribute to which it refers to. By contrast, attribute values can be often split into several words, where each of them is associated with a different pseudorandom number. Therefore, the encryption of these numbers are placed as content of an additional subelement, called `Numbers`, inside the `Node-Info` element.

By having the query template users are able to generate the authorized view of the structure of the document to be inquired, which can be exploited to formulate queries.

4.2 Query generation

In proposed framework, we assume that users submit queries through XPath expressions. XPath allows one to traverse the graph structure of an XML document and to select specific portions on the document according to some properties, such as the type of the elements, or specified content-based conditions. In general, an XPath expression consists of a *location path*, that allows one to select a set of nodes from target documents, which in turn consists of one or more *location steps*, separated among each other by a slash. A location step contains: an *axis*, specifying the tree relationships between the nodes selected by the location step and the current node (e.g., ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self); a *node test*, used to identify a node within an axis, by specifying a node type or the node name (e.g., text(), node()); and zero or more *predicates*, placed inside square brackets, used to further refine the

[8] By completeness we mean that a user receives all document portions he/she is authorized to see according to the owner's access control policies.

[9] We assume that tagnames and attribute names are shorten than $L_{\mathbf{W}}$, i.e., the max length, and thus are treated as a unique word. Therefore, they are associated with a unique pseudorandom number.

set of nodes selected by the location step (e.g., [@Price='30']). By means of predicates an XPath expression can specify conditions on attributes through comparison operators (e.g., $<,>,=$), as well as conditions on textual data, in that it is possible to retrieve XML nodes containing a specified keyword (supported by the *contains()* function).

In order to submit a query to a publisher, the first step that a user has to perform is to download the query template of the requested document from the owner site. By decrypting the query template, the client can locally generate the structure of the authorized view, which can be displayed to the user in order to formulate XPath queries on it. Such queries should then be translated by the client into one or more XPath expressions that could be evaluated by publishers on the encrypted XML documents. In order to do that there are two main transformations to which each location step of a user XPath expression must undergo before being submitted to a publisher. The first implies to cipher the tagnames specified in the node test of the location steps with the proper keys. Note that according to the well-formed encryption, nodes with the same name could be ciphered with different keys, based on the access control policies applied to them. Thus, the ciphering of a location step does not always return a unique value, which implies that for each user's query the client could generate more ciphered queries. The second transformation implies the translation of the query conditions in terms of partition ids and/or encrypted keywords. This is applied to each predicate composing the input XPath expression. If the predicate contains comparison operators (e.g., $<,>,=$), the client substitutes the values appearing in the predicate with the id of the partition to which it belongs to. This is done by using the information about the partitioning functions obtained during the subscription phase. By contrast, if the predicate exploits the contains() function, the client translates the condition by replacing the searched keyword W_j, with the corresponding encrypted word. We recall that according to the adopted scheme, for searching a keyword W_j publishers must be provided with its encryption and with key K_j, which must be sent by the client to publishers together with the submitted query.

To better understand query processing, let us consider an example. In particular, consider an XML document modelling a CD catalog, where a different `CD` element is inserted for each different CD in the catalog. The `CD` element contains an attribute, i.e., `Price`, storing the CD's price and two subelements, i.e., `Title` and `Author`, containing the CD's title and author, respectively. Let us suppose moreover that on this document two different access control policies apply, namely P_1 and P_2. The first authorizes all users to access all the nodes excepts for `Price` attributes. By contrast, P_2 grants store-staff users the access to the whole document. The outsourced document is thus encrypted with two different encryption keys, namely K_1 and K_2. K_1 is associated with policy configuration consisting of both P_1 and P_2 and encrypts all the nodes except for `Price` attributes. K_2 corresponds to policy P_2 and encrypts only the `Price` attributes. A store-staff member u receives by the owner both K_1 and K_2. Suppose now that u wants to submit the following query: `/CD`[@`Price`=30K]/`Title`[contains(.,'Overture')], which returns all the CDs having price equal to 30K and the keyword *Overture* in the title. Before submitting this query the client transforms it, by, at first, ciphering the tagname in each location step obtaining thus the following expression: /C_{K_1}(`CD`) [@C_{K_2}(`Price`)='30K']/C_{K_1}(`Title`)[contains(.,'Overture')]. Note that clients can easily cipher a tagname by simply extracting from the query template the corresponding pseudorandom number. Then, each condition specified in the predicates is translated. This is done by considering the query processing information of the node on which the predicate is specified. Let us for instance consider the predicate "contains(.,'Overture')" on element `Title`. According to the proposed strategy this can be evaluated by searching a ciphered word among those contained in the query processing information of the `Title` element that matches E_{K_1}('Overture'). Thus, the predicate that should be evaluated by the publisher is: /C_{K_1}(`CD`)/C_{K_1}(`Title`)/`Sec-Info` /`Node-Info`/`Query-Info`[contains(.,'$E_{K_1}(Overture)$')].[10] Furthermore, we have to note that in the proposed encoding the ciphered name of an attribute *a* is stored into the `Name` attribute of a `Node-Info` subelement of `Attributes`, which is contained into the `Sec-Info` element associated with the element *e* to which *a* belongs to. This implies that in our example conditions on price should be verified against the query processing information contained into the `Node-Info` element, whose `Name` attribute is equal to C_{K_2}(`Price`). Thus, predicates on the price attributes should be evaluated by the publisher as: /C_{K_1}(`CD`)//`Attributes`/ `Node-Info`[@`Name`=C_{K_2}(`Price`)/`Query-Info` [contains(.,'PF(30')], where PF(30) returns the id of the partition to which the value 30 belongs to. The resulting XPath expression generated by the client is thus: "/C_{K_1}(`CD`)/ C_{K_1}(`Title`)/`Sec-Info`/`Node-Info`/`Query-Info`[contains(.,'$E_{K_1}(Overture)$')] AND /C_{K_1}(`CD`)//`Attributes`/`Node-Info`[@`Name`=C_{K_2}(`Price`)/`Query-Info` [contains(.,'PF(30')]".

4.3 Decryption of query answers

Once the query has been evaluated, the publisher returns to client the encrypted nodes identified by the submitted XPath expressions. According to the adopted document encryption strategy, the client needs to perform two different decryption processes in order to decrypt the obtained nodes: one for tagnames, attribute names and values, and the other for decrypting element contents.

Decryption of tagnames, attribute names and values. Tagnames, attribute names and values are ciphered according to Song et al.'s scheme (cfr. Section 2.2). Given a ciphered word C_j, in order to extract from it the encrypted word $E_K(W_j)$, and thus to decrypt it, the client must be provided with the corresponding pseudorandom numbers. Such numbers are retrieved by the client from the query template (cfr. Section 4.1). By decrypting the pseudorandom numbers, the client is thus able to decrypt the ciphered tagnames, attribute names, and attribute values.

Decryption of element contents. Element contents are encrypted by means of a symmetric encryption algorithm. Therefore, the decryption of element content is simply performed by first retrieving the encryption key associated with the element from the directory server, and then by executing the proper symmetric decryption algorithm.

[10] Note that publishers implement a different contains() function wrt XPath parsers, by, however, preserving the semantics.

5 Conclusions

In this paper, we have proposed a method based on cryptographic techniques for confidentiality enforcement on outsourced XML data. Main benefits of the proposed solution are that it does not make any assumption on the trustworthiness of publishers and it is robust to data dictionary attacks both at the schema and document content level. In the paper, besides illustrating the cryptographic schemes, we show by an example how client-side query processing takes place. Due to space limitations, we have not considered other important issues related to key management, for instance those related to the number of keys that need to be managed. To limit the number of keys we use an hierarchical key management scheme similar to the one proposed by us for temporal access control policies [3], which requires to permanently store a number of keys linear in the number of access control policies.

We are currently implementing the proposed strategies to test the performance of the system in different environments. Moreover, we are currently investigating techniques to efficiently manage updates to policies and documents, that would require a partial re encryption of documents as well as an update of the related security information. In particular, in order to efficiently manage update operations, we plan to adopt a strategy similar to the one in [5] that incrementally maintains document encryptions, by changing all and only those portions which are really affected by the administrative operation, without the need of re-encrypting the document from scratch.

References

1. E.Bertino, B.Carminati, E.Ferrari, B. Thuraisingham, A. Gupta. Selective and Authentic Third-Party Distribution of XML Documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(10):1263–1278, 2004.
2. E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):290–331, 2002.
3. E. Bertino, B. Carminati, and E. Ferrari. A Temporal Key Management Scheme for Broadcasting XML Documents, In Proc. of the *9th ACM Conference on Computer and Communications Security (CCS'02)*, Washington, November, 2002.
4. B. Carminati, E. Ferrari, and E. Bertino. Securing XML Data in Third-Party Distribution Systems. In Proc. of the *ACM Fourteenth Conference on Information and Knowledge Management (CIKM'05)*, Bremen, Germany, November, 2005.
5. B. Carminati, E.Ferrari. Management of Access Control Policies for XML Document Sources. *International Journal of Information Security*, 1(4):236–260, 2003, Springer.
6. H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database Service Provider Model. In Proc. of the *ACM SIGMOD 2002*, Madison, WI, USA, June 2002.
7. H. Hacigumus, B. R. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In Proc. of the *9th International Conference on Database Systems for Advanced Applications*, Jeju Island, Korea, March 2004.
8. R.C. Merkle. A Certified Digital Signature. In *Advances in Cryptology-Crypto '89*, 1989.
9. G. Salton and M. McGill. *Introduction to Modern Information Retrival*. McGraw-Hill, 1983.
10. D. X. Song, D. Wagner and A. Perrig. Practical Techniques for Searches on Encrypted Data, In Proc. of the *IEEE Symposium on Security and Privacy*, Oakland, California, 2000.
11. World Wide Web Consortium. `http://www.w3.org`.

Query Translation for XPath-based Security Views

Roel Vercammen[⋆], Jan Hidders, and Jan Paredaens

University of Antwerp

Abstract. Since XML is used as a storage format in an increasing number of applications, security has become an important issue in XML databases. One aspect of security is restricting access to data by certain users. This can, for example, be achieved by means of access rules or XML security views, which define projections over XML documents. The usage of security views avoids information leakage that may occur when we use certain access rules. XML views can be implemented by materialized views, but materialization and maintenance of views may cause considerable overhead. Therefore, we study translations from queries on views to equivalent queries on the original XML documents, assuming both the security views and the queries are specified by XPath expressions. Especially, we investigate which XPath fragments are closed under the composition of a view and a query.

1 Introduction

Access control mechanisms are essential for database systems used to store and share sensitive information. XML is used in an increasing number of applications, including those handling confidential information. As a consequence, some standards for XML access control have already emerged, such as XACL [11] and XACML [9]. Furthermore, several approaches for XML access control mechanisms have been proposed in the literature [6, 13, 4]. In most of these approaches, the policies are specified at the DTD level. Fundulaki and Marx developed a framework to compare XML access control mechanisms in terms of XPath [8]. Query answering that incorporates these access control policies can, for example, be performed by computing some (materialized) security view [14, 12] and then evaluating the query against this security view. This ensures that no information is exposed that is not supposed to be seen by the user, since the query is evaluated against an XML tree that contains exactly the information the user is allowed to see. However, the materialization of views causes an overhead that might be avoided if we can translate queries on the view to equivalent queries on the original data, without leaking information on "hidden" nodes [6, 2].

In this paper, we will not introduce a new XML access control mechanism, but instead we assume that security views are defined by path expressions p such that access to a node is never granted, except when it is the root node or in the result of p. The obtained XML security views are similar to those of [6] and [12], but we specify them by means of path expressions instead of annotated DTDs. We investigate how to translate queriesst on views to equivalent queries on the original data. Since it is known that

⋆ Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 33581.

some XPath fragments can be evaluated very efficiently [10], we look at a number of XPath fragments to see which of these fragments are closed under the composition of a view and a query.

The rest of the paper is structured as follows. In Section 2 we introduce our XPath-based security views and some preliminary notions. In Section 3 we study the problem of translating queries on XML views to queries on the original (XML) data. We then use these results in Section 4 to examine which XPath fragments are closed under the composition of a view and a query. Finally, we compare our approach to existing query translation mechanisms for queries on XML views in Section 5 and conclude the paper in Section 6.

2 Preliminaries

In this section we introduce some preliminary notions that are used in the rest of our paper. First, we define the data model and the query language that we use in the theoretical exploration of this paper. Next, we introduce our XPath-based security views. Finally, we define the fragments that we investigate.

2.1 Data Model

Our data model is a simplification and abstraction of the full XML data model [7] and restricts itself to the element nodes. First of all, we postulate an infinite set of tag names Σ and an infinite set of nodes $\mathscr{N}$.

Definition 1 (XML Tree). *An* XML tree *is a tuple $T = (N, \lhd, r, \lambda, \prec)$ such that $(N, \lhd, r)$ is a rooted tree where $N \subset \mathscr{N}$ is a finite set of* nodes, *$\lhd$ is the* parent-child relationship, *r is the root, $\lambda : N \rightarrow \Sigma$ labels nodes with their tag name and $\prec$ is a strict total order*[1] *over N that represents the* document order *and defines a pre-order tree-walk, i.e., (1) every child is smaller than its parent, and (2) if two nodes are siblings then all descendants of the smaller sibling are smaller than the larger sibling*

In the following we let $\rhd$ denote the inverse relation of $\lhd$, $\lhd^+$ and $\rhd^+$ the transitive closure of resp. $\lhd$ and $\rhd$, and $\lhd^*$ and $\rhd^*$ the transitive and reflexive closure of resp. $\lhd$ and $\rhd$. The set of all XML trees is denoted by $\mathscr{T}$.

2.2 XPath Queries

We now define the set of XPath expressions we consider. We use a syntax in the style of [1] that abstracts from the official syntax [3] and is more suitable for formal presentations. The largest fragment of XPath that we study in this paper, called $\mathscr{P}$, is defined by the following abstract grammar:

$$\begin{aligned} p ::= {} & \varepsilon \mid \Uparrow \mid l \mid \downarrow \mid \uparrow \mid \downarrow^* \mid \downarrow^+ \mid \uparrow^* \mid \uparrow^+ \mid \leftarrow^+ \mid \rightarrow^+ \mid \twoheadleftarrow \mid \twoheadrightarrow \mid \\ & p/p \mid p[p] \mid p \cap p \mid p \cup p \mid p - p \end{aligned}$$

[1] A strict total order is a binary relation that is irreflexive, transitive and total.

where ε represents the empty path expression or *self* axis, $l \in \Sigma$ denotes a label test, $\uparrow$ and $\downarrow$ represent the *parent* and *child* axis, $\uparrow^*$, $\uparrow^+$, $\downarrow^*$ and $\downarrow^+$ represent the *ancestor-or-self*, *ancestor*, *descendant-or-self* and *descendant* axis, $\leftarrow^+$ and $\rightarrow^+$ represent the *preceding-sibling* and *following-sibling* axis, $\twoheadrightarrow$ and $\twoheadleftarrow$ represent the *following* and *preceding* axis, $\Uparrow$ represents the document root, p_1/p_2 represents the concatenation of p_1 and p_2, $p_1[p_2]$ represents a path p_1 with a *predicate* p_2 and finally $\cap$, $\cup$ and $-$ represent the set intersection, set union and set difference. For disambiguation, parentheses are added and the concatenation is assumed to have the highest precedence. The label tests of the form $l \in \Sigma$ behave as if they follow the *self* axis. This means that `a/b` corresponds to the conventional XPath expression `self::a/self::b` and *not* to the expression `child::a/child::b` as is the case for the so-called abbreviated XPath syntax. Based on [5] and similar to [1] we define the semantics as follows:

Definition 2 (XPath Semantics). *Given an XML tree $T = (N, \lhd, r, \lambda, \prec)$ we define the* semantics *of a path expression p, $\llbracket p \rrbracket_T \subseteq N \times N$ as follows:*

$$\begin{array}{llll}
\llbracket \Uparrow \rrbracket_T & = \{(n,n') \mid n' = r\} & & \\
\llbracket \uparrow \rrbracket_T & = \rhd & \llbracket \downarrow \rrbracket_T & = \lhd \\
\llbracket \uparrow^* \rrbracket_T & = \rhd^* & \llbracket \downarrow^* \rrbracket_T & = \lhd^* \\
\llbracket \uparrow^+ \rrbracket_T & = \rhd^+ & \llbracket \downarrow^+ \rrbracket_T & = \lhd^+ \\
\llbracket \twoheadleftarrow \rrbracket_T & = \succ - \rhd^* & \llbracket \twoheadleftarrow \rrbracket_T & = \prec - \lhd^* \\
\llbracket \leftarrow^+ \rrbracket_T & = \succ \cap (\rhd \circ \lhd) & \llbracket \rightarrow^+ \rrbracket_T & = \prec \cap (\rhd \circ \lhd) \\
\llbracket \varepsilon \rrbracket_T & = \{(n,n') \mid n = n'\} & \llbracket l \rrbracket_T & = \{(n,n') \mid n = n' \wedge \lambda(n) = l\} \\
\llbracket p_1/p_2 \rrbracket_T & = \llbracket p_1 \rrbracket_T \circ \llbracket p_2 \rrbracket_T & \llbracket p_1 \cap p_2 \rrbracket_T & = \llbracket p_1 \rrbracket_T \cap \llbracket p_2 \rrbracket_T \\
\llbracket p_1 \cup p_2 \rrbracket_T & = \llbracket p_1 \rrbracket_T \cup \llbracket p_2 \rrbracket_T & \llbracket p_1 - p_2 \rrbracket_T & = \llbracket p_1 \rrbracket_T - \llbracket p_2 \rrbracket_T \\
\llbracket p_1[p_2] \rrbracket_T & = \{(n,n') \mid (n,n') \in \llbracket p_1 \rrbracket_T \wedge \exists n'' : (n',n'') \in \llbracket p_2 \rrbracket_T\} & &
\end{array}$$

Note that "$\circ$" denotes the concatenation of binary relations (and therefore also functions), i.e., $(x,y) \in (f \circ g) \Leftrightarrow \exists z : (x,z) \in f \wedge (z,y) \in g$. This is the reverse of the usual semantics. The length of a path expression p is denoted by $|p|$ and equals the size of the abstract syntax tree of p. Let p be a path expression. We define p^k as the concatenation of k times p, i.e., $p^0 = \varepsilon$ and $p^{n+1} = p^n/p$.

Definition 3 (Query). *Let p be a path expression. The query $\mathscr{Q}[p]$ is a function $\mathscr{T} \rightarrow 2^{\mathscr{N}}$, defined as follows: $\forall T = (N, \lhd, r, \lambda, \prec) : (n \in \mathscr{Q}[p](T) \Leftrightarrow (r,n) \in \llbracket p \rrbracket_T)$*

Note that $\forall T \in \mathscr{T} : \llbracket p_1 \rrbracket_T = \llbracket p_2 \rrbracket_T$ implies $\mathscr{Q}[p_1] = \mathscr{Q}[p_2]$, but the reverse does not necessarily hold. For example, we know that $\mathscr{Q}[\downarrow/\uparrow^+] = \mathscr{Q}[\varepsilon[\downarrow]]$, but for many XML trees T, $\llbracket \downarrow/\uparrow^+ \rrbracket_T \neq \llbracket \varepsilon \rrbracket_T$.

2.3 XPath-based Security Views

The XML views that we consider are similar to the XML security views of [6, 12]. However, we define security views by means of path expressions instead of annotating the DTD. Informally, a view defined by a path expression p maps an XML tree, called *input tree*, to an XML tree, called *view tree*, such that the view tree is the projection of the input tree on the nodes selected by p and the root of the input tree. We always include the root of the input tree in order to ensure that the projection yields a valid XML tree instead of a forest.

Definition 4 (View). *Let p be a path expression. The view $\mathscr{V}[p]$ is a function $\mathscr{T} \rightarrow \mathscr{T}$, defined as follows:* $\forall T_1 = (N_1, \lhd_1, r_1, \lambda_1, \prec_1), T_2 = (N_2, \lhd_2, r_2, \lambda_2, \prec_2) : \mathscr{V}[p](T_1) = T_2 \Leftrightarrow$

– $N_2 = \{n | (n = r_1) \vee ((r_1, n) \in [\![p]\!]_{T_1})\}$
– $\lhd_2 = \{(m,n) | (m,n \in N_2) \wedge (m \lhd_1^+ n) \wedge (\nexists n' \in N_2 : (m \lhd_1^+ n') \wedge (n' \lhd_1^+ n))\}$
– $r_2 = r_1$
– *λ_2 and $\prec_2$ are restrictions of λ_1 and $\prec_1$ to N_2*

Example 1. A governmental organization has to check hospitals and the treatments that are performed by their doctors. For privacy reasons, hospitals are not allowed to transfer *any* information on their patients to this institute. The doctors of this hospital, however, have internally organized the information on treatments by collecting them per patient. Suppose the hospital database is the left tree in Fig. 1 and the government wants the data in the form of the right tree in this figure. We can obtain the right tree using an XPath-based security view, more precisely the view $\mathscr{V}[\downarrow^+/\texttt{Doctor}/(\varepsilon \cup \downarrow^+/\texttt{Treatment})]$ transforms input trees of the left form to view trees of the right form.

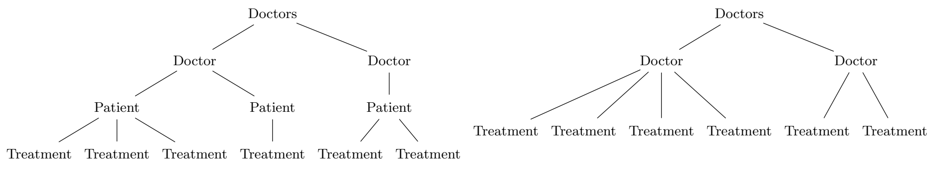


Fig. 1. Input and View Tree of Example 2.3

Note that the semantics of path expressions on the view tree in terms of the input tree differs from the semantics of the same path expression on the input tree. For example, in Fig. 1 a ‘Treatment’ node is a child of a ‘Doctor’ node in the view, while this is not true in the input tree. However, for some axes a it holds that they are “robust under view definition”, i.e., $[\![a]\!]_{\mathscr{V}[p](T)} \subseteq [\![a]\!]_T$. The robust axes are $\varepsilon, \uparrow^*, \uparrow^+, \downarrow^*, \downarrow^+, \twoheadleftarrow$, and $\twoheadrightarrow$. As we show in Section 3, these axes can easily be translated. Moreover, we can express all other axes in terms of these axes: $[\![\downarrow]\!]_T = [\![\downarrow^+ - \downarrow^+/\downarrow^+]\!]_T$, $[\![\uparrow]\!]_T = [\![\uparrow^+ - \uparrow^+/\uparrow^+]\!]_T$, $[\![\rightarrow^+]\!]_T = [\![(\uparrow^+ - \uparrow^+/\uparrow^+)/(\downarrow^+ - \downarrow^+/\downarrow^+) \cap \twoheadrightarrow]\!]_T$, and $[\![\leftarrow^+]\!]_T = [\![(\uparrow^+ - \uparrow^+/\uparrow^+)/(\downarrow^+ - \downarrow^+/\downarrow^+) \cap \twoheadleftarrow]\!]_T$.
In some query translations, we first transform path expressions to equivalent expressions only containing robust axes. Since none of our fragments contain the following and preceding axes, we afterwards remove them using following equalities: $[\![\twoheadrightarrow]\!]_T = [\![\uparrow^*/\rightarrow^+/\downarrow^*]\!]_T$ and $[\![\twoheadleftarrow]\!]_T = [\![\uparrow^*/\leftarrow^+/\downarrow^*]\!]_T$.

2.4 XPath Fragments

We now define the XPath fragments that we study in this paper and discuss some of their properties. These fragments are inspired by the fragments introduced in [1], but we have

added sibling axes, intersection, union and set difference. Furthermore, their label tests l correspond to $\downarrow/l$ in our XPath model. Our fragments are defined by the axes that can occur in path expressions and the different operations we can use to combine two path expressions to a new path expression. We consider two groups of fragments. One is defined by a base fragment $\mathscr{X}$ and loosely corresponds to the fragments introduced in [1]; the other is defined by a base fragment $\mathscr{A}$ which is based on the abbreviated syntax [3].
The fragment $\mathscr{X}$ is defined as

$$p ::= \varepsilon \mid \Uparrow \mid l \mid \downarrow \mid p/p.$$

We can extend this fragment by adding the parent axis ($\uparrow$), adding the sibling axes ($\leftarrow^{+}$ and $\rightarrow^{+}$), and adding the transitive and reflexive closure of axes (i.e., adding $\downarrow^{*}$ and if $\uparrow$ is in the fragment then also $\uparrow^{*}$). The three possible extensions can be combined arbitrarily and are respectively denoted by superscripts $\uparrow$, $\leftrightarrow$, and r.
The fragment $\mathscr{A}$ is defined as

$$p ::= \varepsilon \mid \Uparrow \mid \downarrow \mid \downarrow/l \mid \downarrow^{*} \mid \uparrow \mid p/p.$$

All previous fragments can be extended with predicates ([ ]), set intersection ($\cap$), set union ($\cup$), and set difference ($-$). The addition of these extensions is denoted by subscripts.

Some fragments F contain path expressions that are equivalent to path expressions that are not in F. If the addition of a certain operation o to a fragment F does not increase the expressive power of path expressions defined in F then we say that o can be expressed in F. Furthermore, for some fragments F it holds that we cannot in general express an operation o in F, but we can express o if we assume that all path expressions are evaluated against the root. We then say that o can be expressed in queries of F. We now give some expressibility results for the XPath fragments that we have just defined.

Lemma 1. *The following expressibility properties hold for queries, i.e., path expressions evaluated against the root of a tree:*

1. *The union of two path expressions can be expressed in all fragments that can express the set difference and the descendant-or-self axis.*
2. *The intersection of two path expressions can be expressed in all fragments that can express the set difference.*
3. *Predicates can be expressed in all fragments that can express intersection.*
4. *Parent, ancestor and ancestor-or-self axes can be expressed in all fragments that can express intersection and descendant-or-self axes.*

The first two properties also hold for path expressions in general.

Proof. (Sketch)

1. This follows from $[\![p_1 \cup p_2]\!]_T = [\![(\Uparrow/\downarrow^{*}) - ((\Uparrow/\downarrow^{*}) - p_1 - p_2)]\!]_T$.
2. This follows from $[\![p_1 \cap p_2]\!]_T = [\![p_1 - (p_1 - p_2)]\!]_T$.

3. We can define a function $e : \mathscr{P} \times \mathscr{P} \to \mathscr{P}$ such that $\mathscr{Q}[p_c/p] = \mathscr{Q}[p_c/e(p,p_c)]$ and its result does not contain predicates if the second argument does not contain predicates. For predicate operations the mapping is defined by $e(p_1[p_2],p_c) = e(p_1,p_c)/(\varepsilon \cap e(p_2,p_c/e(p_1,p_c))/\Uparrow/p_c/e(p_1,p_c))$. For all other operations the definition is straightforward, e.g., $e(p_1 \cup p_2,p_c) = e(p_1,p_c) \cup e(p_2,p_c)$, and $e(p_1/p_2,p_c) = e(p_1,p_c)/e(p_2,p_c/e(p_1,p_c))$.
4. Similar to the previous part of this proof, we can define a function $e : \mathscr{P} \times \mathscr{P} \to \mathscr{P}$ such that $\mathscr{Q}[p_c/p] = \mathscr{Q}[p_c/e(p,p_c)]$ and $e(p,p_c)$ does not contain $\uparrow$, $\uparrow^+$ or $\uparrow^*$ if the second argument does not contain these axes. The mapping for $\uparrow^*$ is defined by $e(\uparrow^*,p_c) = \Uparrow/\downarrow^*[\downarrow^* \cap \Uparrow/p_c]$ and similar mappings can be defined for $\uparrow$ and $\uparrow^+$. The mapping of predicate operations differs from part 3: $e(p_1[p_2],p_c) = e(p_1,p_c)[e(p_2,p_c/e(p_1,p_c)]$. Since predicates can be expressed using intersection, we only need $\cap$ and $\downarrow^*$ axes to express $\uparrow$, $\uparrow^+$, and $\uparrow^*$.

□

From the previous lemma follows that $\mathscr{P}$ has the same expressive power as $\mathscr{X}_{-}^{r,\leftrightarrow}$. We conclude this section by showing that for some queries $\mathscr{Q}[p]$ we know that all nodes in $\mathscr{Q}[p](T)$ are on the same depth in T.

Lemma 2. *For all path expressions $p \in \mathscr{X}_{[\,],\cap,-}^{\uparrow,\leftrightarrow}$ it holds that for all XML trees T all nodes in the result of $\mathscr{Q}[p](T)$ are on the same level $d(p,0)$, inductively defined as follows:*

$d(\Uparrow,n) = 0$	$d(\varepsilon,n) = n$	$d(l,n) = n$
$d(\downarrow,n) = n+1$	$d(\uparrow,n) = n-1$	$d(p_1/p_2,n) = d(p_2,d(p_1,n))$
$d(\leftarrow^+,n) = n$	$d(\rightarrow^+,n) = n$	$d(p_1[p_2],n) = d(p_1,n)$
$d(p_1 \cap p_2,n) = d(p_1,n)$	$d(p_1 - p_2,n) = d(p_1,n)$	

Proof. (Sketch) For all $p \in \mathscr{X}_{[\,],\cap,-}^{\uparrow,\leftrightarrow}$ it can be shown by induction on the length of p that if n_1 is a node in T at depth n and $(n_1,n_2) \in [\![p]\!]_T$ then n_2 is a node at depth $d(p,n)$ in T. □

3 Composing Views and Queries

In this section we study the problem of composing a view and a query to a new query on the input tree that is equivalent to the query on the view tree. We propose two translations, one that can be used to translate path expressions on view trees to path expressions on input trees and one that can only be used to translate queries on view trees to queries on input trees.

Definition 5. *Let p be a path expression. The function $\tau_p : \mathscr{P} \to \mathscr{P}$ is defined as follows:*

$\tau_p(\Uparrow) = \Uparrow$	$\tau_p(\varepsilon) = \varepsilon$
$\tau_p(l) = l$	$\tau_p(q_1/q_2) = \tau_p(q_1)/\tau_p(q_2)$
$\tau_p(q_1[q_2]) = \tau_p(q_1)[\tau_p(q_2)]$	$\tau_p(q_1 \cap q_2) = \tau_p(q_1) \cap \tau_p(q_2)$
$\tau_p(q_1 \cup q_2) = \tau_p(q_1) \cup \tau_p(q_2)$	$\tau_p(q_1 - q_2) = \tau_p(q_1) - \tau_p(q_2)$
$\tau_p(\downarrow^*) = \downarrow^* \cap \Uparrow/(p \cup \varepsilon)$	$\tau_p(\uparrow^*) = \uparrow^* \cap \Uparrow/(p \cup \varepsilon)$
$\tau_p(\twoheadrightarrow) = \twoheadrightarrow \cap \Uparrow/(p \cup \varepsilon)$	$\tau_p(\twoheadleftarrow) = \twoheadleftarrow \cap \Uparrow/(p \cup \varepsilon)$

We now show that this definition can be used to translate path expressions on view trees to path expressions on input trees.

Lemma 3. *Let p, q be path expressions. For all XML trees $T = (N, \lhd, r, \lambda, \prec)$ and $T' = (N', \lhd', r, \lambda', \prec')$ it holds that if $\mathscr{V}[p](T) = T'$ then $[\![\tau_p(q)]\!]_T \cap (N' \times N) = [\![q]\!]_{T'}$ and therefore $\mathscr{V}[p] \circ \mathscr{Q}[q] = \mathscr{Q}[\tau_p(q)]$. Furthermore, $|\tau_p(q)| = O(|p| \times |q|)$*

Proof. (Sketch) This lemma can be shown by induction on $|q|$. Note that since $\uparrow^*, \downarrow^*, \twoheadleftarrow$ and $\twoheadrightarrow$ are robust axes, they can be translated by following the same axis and restricting the result nodes to nodes in T', which are the result nodes of $\Uparrow/(p \cup \varepsilon)$. Finally, $|\tau_p(q)| = O(|p| \times |q|)$, since each of the $|q|$ steps is translated into a path expression of size $O(|p|)$. □

Using the previous result, we can translate q to $\tau_p(q)$ such that q evaluated against a node n in the view tree and $\tau_p(q)$ evaluated against a node n' in the input tree always return the same result when $n' = n$. This property might be too strong, since for some fragments it can be impossible to find such a translation, but we can find a translation for path expressions evaluated against the root.

Definition 6. *Let p, q be path expressions in $\mathscr{X}^{\uparrow,\leftrightarrow}_{[\,],\cap,-}$. The function $\rho_p : \mathscr{P} \times \mathbb{N} \to \mathscr{P}$ is defined as follows:*

$\rho_p(q, n)$	$= q$	*(if $n \in \{0, 1\}$ and $q \in \{\Uparrow, \varepsilon\} \cup \Sigma$)*	
$\rho_p(\downarrow, 0)$	$= p$	*(if $d(p, 0) > 0$)*	
$\rho_p(\uparrow, 1)$	$= \Uparrow$	*(if $d(p, 0) > 0$)*	
$\rho_p(\leftarrow^+, n)$	$= \Uparrow/p \cap (\bigcup_{l=0}^{d(p,0)} \uparrow^l/\leftarrow^+/\downarrow^l)$	$\rho_p(\rightarrow^+, n)$	$= \Uparrow/p \cap (\bigcup_{l=0}^{d(p,0)} \uparrow^l/\rightarrow^+/\downarrow^l)$
$\rho_p(q_1/q_2, n)$	$= \rho_p(q_1, n)/\rho_p(q_2, d(q_1, n))$	$\rho_p(q_1[q_2], n)$	$= \rho_p(q_1, n)[\rho_p(q_2, d(q_1, n))]$
$\rho_p(q_1 \cap q_2, n)$	$= \rho_p(q_1, n) \cap \rho_p(q_2, n)$	$\rho_p(q_1 - q_2, n)$	$= \rho_p(q_1, n) - \rho_p(q_2, n)$

In the cases not covered by the above equations, $\rho_p(q, n) = p_\emptyset$, where $p_\emptyset$ is a shorthand for a path expression that is not satisfiable, e.g., a/b with $a, b \in \Sigma$ and $a \neq b$.

Lemma 4. *If $p, q \in \mathscr{X}^{\uparrow,\leftrightarrow}_{[\,],\cap,-}$ then $\mathscr{Q}[\rho_p(q, 0)] = \mathscr{V}[p] \circ \mathscr{Q}[q]$. Furthermore, $|\rho_p(q, 0)| = O(|p|^2 \times |q|)$.*

Proof. (Sketch) We show by induction on $|q|$ that for all $p, q \in \mathscr{X}^{\uparrow,\leftrightarrow}_{[\,],\cap,-}$ it holds that if n_1 is a node at depth n in $\mathscr{V}[p](T)$ then $(n_1, n_2) \in [\![q]\!]_{\mathscr{V}[p](T)}$ iff $(n_1, n_2) \in [\![\rho_p(q, n)]\!]_T$. Afterwards, it clearly holds that $\mathscr{Q}[\rho_p(q, 0)] = \mathscr{V}[p] \circ \mathscr{Q}[q]$, since a query is always evaluated from the root node. From Lemma 2 we know that all nodes selected by p are on the same level and hence the view tree does not contain nodes at depth 2 or more. Consequently, we can sometimes determine statically whether a certain operation jumps out of the view tree, yielding an empty result set. If $d(p, 0) > 0$ then the set of nodes on level 1 in the view tree is $\mathscr{Q}[p](T)$. Hence following $\downarrow$ from level 0 in the view tree corresponds to evaluating p against the root in the input tree and following $\uparrow$ from level 1 corresponds to $\Uparrow$. The evaluation of $\rightarrow^+$ in the view tree corresponds to getting all following nodes on level $d(p, 0)$ in the input tree and checking whether they are in $\mathscr{Q}[p](T)$. The translation of $\leftarrow^+$ is similar and the translation for the other operations is straightforward and similar to τ_p.

Finally, $|\rho_p(q, 0)| = O(|p|^2 \times |q|)$, since each of the $|q|$ steps is translated into a path expression of size $O(|p|^2)$ (and $O(|p|)$ if q does not contain sibling axes). □

4 Closure of XPath Fragments under View Composition

In this section we investigate for each fragment F, defined in Subsection 2.4, if it is closed under view composition, i.e., $\forall p_1, p_2 \in F : \exists p_3 \in F : \mathscr{V}[p_1] \circ \mathscr{Q}[p_2] = \mathscr{Q}[p_3]$. We first look at fragments without set difference, called positive XPath fragments, and then at fragments with set difference.

4.1 View Composition for Positive XPath Fragments

The following table summarizes which positive fragments are closed under view composition. Each cell in this table denotes one fragment, i.e., the fragment that can be obtained by adding the operations in the column head to the fragment that is in the row head. If a "○" is in a certain cell then this fragment is not closed under view composition, otherwise there is a "•" to denote that the fragment is closed under view composition. Next to each "•" and "○" symbol there is a number that refers to the theorem that shows the result for this fragment.

		$[\,]$	$\cap$	$\cup$	$[\,],\cap$	$[\,],\cup$	$\cap,\cup$	$[\,],\cap,\cup$
$\mathscr{X}$	$\bullet_1$	$\bullet_1$	$\bullet_1$	$\circ_2$	$\bullet_1$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\uparrow}$	$\bullet_1$	$\bullet_1$	$\bullet_1$	$\circ_2$	$\bullet_1$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\leftrightarrow}$	$\circ_4$	$\circ_4$	$\circ_4$	$\circ_2$	$\circ_4$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{r}$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_2$	$\circ_3$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\uparrow,\leftrightarrow}$	$\circ_4$	$\circ_4$	$\circ_4$	$\circ_2$	$\circ_4$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\leftrightarrow,r}$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_2$	$\circ_3$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\uparrow,r}$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_2$	$\circ_3$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{X}^{\uparrow,\leftrightarrow,r}$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_2$	$\circ_3$	$\circ_2$	$\circ_2$	$\circ_2$
$\mathscr{A}$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_3$	$\circ_3$

Theorem 1. *All fragments from $\mathscr{X}$ to $\mathscr{X}^{\uparrow}_{[\,],\cap}$ are closed under view composition.*

Proof. (Sketch) From Lemma 4 we know that if p, q in $\mathscr{X}^{\uparrow}_{[\,],\cap}$ then $\mathscr{Q}[\rho_p(q,0)] = \mathscr{V}[p] \circ \mathscr{Q}[q]$. Note that in ρ_p parent axes, predicates and intersection only occur in the resulting path expression iff they occur in q or p. □

Lemma 5. *Let $p \in \mathscr{X}^{\uparrow,\leftrightarrow,r}_{[\,],\cap,\cup}$ and T an XML tree. If T' is T where some nodes are renamed to a new node name that does not occur in p, then $[\![p]\!]_{T'} \subseteq [\![p]\!]_T$.*

Proof. (Sketch) We prove this lemma by induction on $|p|$. The semantics of axes in T and T' are the same. The semantics of label tests changes, but for all label tests l that occur in p it holds that $[\![l]\!]_{T'} \subseteq [\![l]\!]_T$. Finally, if $[\![p_1]\!]_{T'} \subseteq [\![p_1]\!]_T$ and $[\![p_2]\!]_{T'} \subseteq [\![p_2]\!]_T$, then for path expressions p of the form $p_1[p_2]$, $p_1 \cap p_2$ or $p_1 \cup p_2$ it clearly holds that $[\![p]\!]_{T'} \subseteq [\![p]\!]_T$. □

Theorem 2. *All fragments from $\mathscr{X}_{\cup}$ to $\mathscr{X}^{\uparrow,\leftrightarrow,r}_{[\,],\cap,\cup}$ are not closed under view composition.*

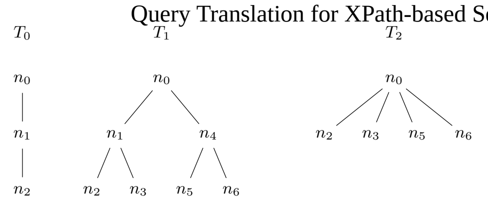


Fig. 2. Counter examples for proofs of Theorems 2, 3, and 4

Proof. (Sketch) Suppose $p \in \mathscr{X}^{\uparrow,\leftrightarrow,r}_{[\,],\cap,\cup}$and $\mathscr{Q}[p] = \mathscr{V}[(\downarrow/a) \cup (\downarrow/\downarrow)] \circ \mathscr{Q}[\downarrow]$. Let T be the tree T_0 shown in Fig. 2 with $\lambda(n_1) =$ "a" and T' be the same XML tree as T except that n_1 has a label which is different from "a" and all labels for which a test occurs in p. From Lemma 5 it follows that $[\![p]\!]_{T'} \subseteq [\![p]\!]_T$. However, $\mathscr{Q}[p](T) = \{n_1\}$ and $\mathscr{Q}[p](T') = \{n_2\}$. □

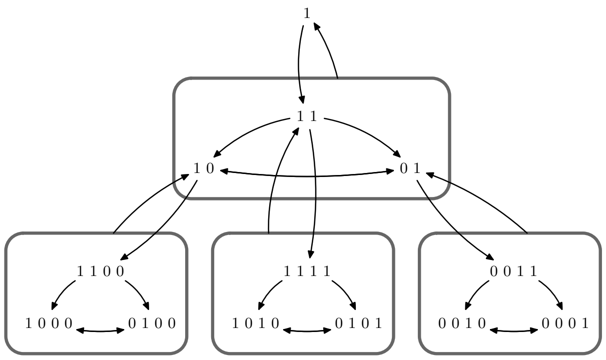
Theorem 3. *All fragments from $\mathscr{X}^r$ to $\mathscr{X}^{\uparrow,\leftrightarrow,r}_{[\,],\cap,\cup}$ and from $\mathscr{A}$ to $\mathscr{A}_{[\,],\cap,\cup}$ are not closed under view composition.*

Proof. (Sketch) Suppose $p \in \mathscr{X}^{\uparrow,\leftrightarrow,r}_{[\,],\cap,\cup}$ and $\mathscr{Q}[p] = \mathscr{V}[\downarrow^*/\downarrow/a] \circ \mathscr{Q}[\downarrow]$. Let T be the tree T_0 shown in Fig. 2 with $\lambda(n_1) = \lambda(n_2) =$ "a" and T' be the same XML tree as T except that n_1 has a label which is different from "a" and all labels for which a test occurs in p. From Lemma 5 it follows that $[\![p]\!]_{T'} \subseteq [\![p]\!]_T$. However, $\mathscr{Q}[p](T) = \{n_1\}$ and $\mathscr{Q}[p](T') = \{n_2\}$. □

Theorem 4. *All fragments from $\mathscr{X}^{\leftrightarrow}$ to $\mathscr{X}^{\uparrow,\leftrightarrow}_{[\,],\cap}$ are not closed under view composition.*

Proof. (Sketch) Suppose $p \in \mathscr{X}^{\uparrow,\leftrightarrow}_{\cap}$ and $\mathscr{Q}[p] = \mathscr{V}[\downarrow/\downarrow] \circ \mathscr{Q}[\downarrow/\rightarrow^+]$. Since both the view and the query do not contain label tests, we may assume that p does not contain label tests. Let T_1 be a tree of the form shown in Fig. 2. The tree T_2 in this figure is obviously $\mathscr{V}[\downarrow/\downarrow](T_1)$ and hence $\mathscr{Q}[p](T_1) = \mathscr{Q}[\downarrow/\rightarrow^+](T_2) = \{n_3, n_5, n_6\}$. From Lemma 2 we know $\mathscr{Q}[p](T_1)$ only contains nodes at depth $d(p,0)$ in T_1. We can encode $\mathscr{Q}[p](T_1)$ as a string of $0's$ and $1's$, where a 0 at position i denotes the absence of, and a 1 the presence of the i^{th} node at level $d(p,0)$ in $\mathscr{Q}[p](T)$. For example, $\mathscr{Q}[p](T_1)$, which is $\{n_3, n_5, n_6\}$, is encoded by 0111.

We show by induction on $|p|$ that $\mathscr{Q}[p](T_1)$ cannot be encoded by 0111. Since queries start from the root, the result of ε is encoded by 1. The following diagram shows all possible "state transitions" of $\uparrow, \downarrow, \leftarrow^+$, and $\rightarrow^+$. Note that we omit transitions to empty results, since these states are sink states.



Finally, the intersection combines two of the states in the previous diagram and, as can easily be verified, goes again to one of the states in this diagram. Hence, the encodings for all possible results of path expressions on T_1 in $\mathscr{X}_{\cap}^{\uparrow,\leftrightarrow}$ (without node tests) are listed in this diagram, which does not contain 0111, so p cannot be expressed in $\mathscr{X}_{\cap}^{\uparrow,\leftrightarrow}$ and by Lemma 1 also not in $\mathscr{X}_{[\,],\cap}^{\uparrow,\leftrightarrow}$. □

4.2 View Composition for Fragments with Set Difference

The following table summarizes which fragments with set difference are closed (●) under view composition and which are not (○).

	-	$[\,],-$	$\cap,-$	$\cup,-$	$[\,],\cap,-$	$[\,],\cup,-$	$\cap,\cup,-$	$[\,],\cap,\cup,-$
$\mathscr{X}$	$\bullet_5$	$\bullet_5$	$\bullet_5$	$\bullet_7$	$\bullet_5$	$\bullet_7$	$\bullet_7$	$\bullet_7$
$\mathscr{X}^{\uparrow}$	$\bullet_5$	$\bullet_5$	$\bullet_5$	$\bullet_7$	$\bullet_5$	$\bullet_7$	$\bullet_7$	$\bullet_7$
$\mathscr{X}^{\leftrightarrow}$	$\bullet_5$	$\bullet_5$	$\bullet_5$	$\bullet_7$	$\bullet_5$	$\bullet_7$	$\bullet_7$	$\bullet_7$
$\mathscr{X}^{r}$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$
$\mathscr{X}^{\uparrow,\leftrightarrow}$	$\bullet_5$	$\bullet_5$	$\bullet_5$	$\bullet_7$	$\bullet_5$	$\bullet_7$	$\bullet_7$	$\bullet_7$
$\mathscr{X}^{\uparrow,r}$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$
$\mathscr{X}^{\leftrightarrow,r}$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$
$\mathscr{X}^{\uparrow,\leftrightarrow,r}$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$
$\mathscr{A}$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$	$\bullet_6$

Theorem 5. *All fragments from $\mathscr{X}_{-}$ to $\mathscr{X}_{[\,],\cap,-}^{\uparrow,\leftrightarrow}$ are closed under view composition.*

Proof. (Sketch) Let $p,q \in \mathscr{X}_{[\,],\cap}^{\uparrow,\leftrightarrow}$. Using ρ_p we can create a query $\mathscr{Q}[\rho_p(q,0)] = \mathscr{V}[p] \circ \mathscr{Q}[q]$. Note that predicates only occur in $\rho_p(q,0)$ iff they occur in q or p. Since we have set difference, by Lemma 1 we can express intersection and hence predicates. We also can express a parent axis in $\mathscr{X}_{-}$, which can be shown by changing $e(\uparrow, p_c)$ of part 4 of the proof of Lemma 1 to $\Uparrow/(\downarrow)^{d(p_c,0)-1}[\downarrow \cap \Uparrow/p_c]$, because all "candidate parents" are at depth $d(p_c,0)-1$. Finally, the translation of the sibling axes can be expressed in $\mathscr{X}_{-}$. For example, $[\![\rho_p(\leftarrow^{+},n)]\!]_T = [\![\Uparrow/p-(\Uparrow/p-\leftarrow^{+}-\uparrow/\leftarrow^{+}/\downarrow-\ldots-(\uparrow)^{d(p,0)}/\leftarrow^{+}/(\downarrow)^{d(p,0)})]\!]_T$ as can be verified. □

Theorem 6. *All fragments from $\mathscr{X}_{-}^{r}$ to $\mathscr{X}_{[\,],\cap,\cup,-}^{\uparrow,\leftrightarrow,r}$, and from $\mathscr{A}_{-}$ to $\mathscr{A}_{[\,],\cap,\cup,-}$ are closed under view composition.*

Proof. (Sketch) We use τ_p, for which we know that $\tau_p(q)$ does not contain sibling axes if they do not occur in p and q. Moreover, from Lemma 1 we know that $\uparrow^*$, $\cup$, $\cap$ and predicates can be expressed using set difference and $\downarrow^*$. □

Theorem 7. *All fragments from $\mathscr{X}_{\cup,-}$ to $\mathscr{X}^{\uparrow,\leftrightarrow}_{[\,],\cap,\cup,-}$ are closed under view composition.*

Proof. (Sketch) We use τ_p to prove this theorem. Since we can express intersection and predicates using set difference, we can eliminate these operations in $\tau_p(q)$. No recursive axes ($\downarrow^*, \uparrow^*$) are allowed in p and hence there is a depth k such that all nodes deeper than k cannot influence the result of q, since they can simply never be in the view. Hence, we can simulate the $\downarrow^*$ axes in $\tau_p(q)$ by $\bigcup_{i=0}^{k} \downarrow^i$ for some k of which the value depends on p. From part 4 of Lemma 1 then follows that we also can simulate the $\uparrow^*$ axes. Finally, $\tau_p(q)$ does not contain sibling axes if they do not occur in p and q. □

4.3 Summary of Results

All fragments with recursive axes, sibling axes, or set union and without set difference are not closed under view composition, while all other fragments are closed. It can easily be verified that for all fragments that are closed under view composition the size of translated queries for $\mathscr{V}[p] \circ \mathscr{Q}[q]$ is $O(|p| \times |q|)$, except for (1) fragments containing sibling axes and no recursive axes, where the size of the translated query is $O(|p|^2 \times |q|)$, due to the translation of sibling axes in ρ_p, and (2) fragments containing set union and set difference, and no recursive axis steps, where the size of the translated query is also $O(|p|^2 \times |q|)$ (see proof of Theorem 7).

5 Related Work

We briefly discuss two existing approaches for translating queries on XML views to queries on the original (XML) data and compare them to our approach.

Fan, Chan, and Garofalakis introduce the notion of XML security views in [6], where they specify views in terms of normalized DTDs. They present a query translation mechanism for their XPath fragment, which more or less corresponds to our fragment $\mathscr{X}^{r}_{[\,],\cup}$, augmented with predicates containing boolean operators ($\wedge, \vee, \neg$) and comparisons of the contents of a node with constant values. They also look at the optimization of the obtained path expressions.

Benedikt and Fundulaki investigate the specification and composition of XML subtree queries [2]. A subtree query is specified by a path expression and is, just like our XPath-based security views, a projection of nodes from an input tree. While in our views intermediate nodes can be hidden, subtree queries show also all descendants and ancestors. It is for example true that if one node in a view is a child of another node in the same view then the former is also a child of the latter in the input tree, which is not necessarily true in our notion of views. More fragments are closed under the composition of subtree queries than under the composition of a view and a query. Note that our notion of views can also be used to express subtree queries: if p is a path expression that specifies a subtree query then this subtree query is equivalent to the view specified by $p/\downarrow^*/\uparrow^*$.

6 Conclusion and Future Work

In this paper we introduce XPath-based security views that define a projection of a tree that only contains the root and the nodes that are selected by an XPath expression. We investigate how to translate XPath queries on such views to XPath queries on the original trees. More specifically, we show which fragments are closed under such a composition of a view and a query. In future work we plan to investigate the translation of path expressions that start from arbitrary nodes in the view. We also plan to include extra knowledge that we can obtain from DTDs. Moreover, we want to see whether a DTD for the view tree can automatically be derived from the DTD of the input tree and the view definition.

References

1. M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *ICDT 2003*, pages 79–95, 2003.
2. M. Benedikt and I. Fundulaki. XML subtree queries: Specification and composition. In *DBPL 2005*, pages 138–153, 2005.
3. A. Berglund, S. Boag, D. Chamberlin, M. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, W3C working draft, 2005. `http://www.w3.org/TR/xpath20`.
4. E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331, 2002.
5. D. Draper, P. Frankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, 2005.
6. W. Fan, C. Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598, 2004.
7. M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model (XDM), 2005. `http://www.w3.org/TR/xpath-datamodel/`.
8. I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *SACMAT 2004*, pages 61–69, 2004.
9. S. Godik and T. Moses, editors. *eXtensible Access Control Markup Language (XACML) Version 1.0.* February 2003.
10. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS 2003*, pages 179–190, San Diego, California, 2003.
11. M. Kudo and S. Hada. XML access control. `http://www.trl.ibm.com/projects/xml/xacl/`.
12. G. Kuper, M. Fabio, and R. Nataliya. Generalized XML security views. In *SACMAT 2005*, pages 77–84, 2005.
13. M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *CCS*, pages 73–84, 2003.
14. A. Stoica and C. Farkas. Secure XML views. In E. Gudes and S. Shenoi, editors, *DBSec*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer, 2002.

Management of executable schema mappings for XML data exchange

Tadeusz Pankowski[1,2]

[1] Institute of Control and Information Engineering,
Poznań University of Technology, Poland
[2] Faculty of Mathematics and Computer Science,
Adam Mickiewicz University, Poznań, Poland
tadeusz.pankowski@put.poznan.pl

Abstract. Executable schema mappings between XML schemas are essential to support numerous data management tasks such as data exchange, data integration and schema evolution. The novelty of this paper consists in a method for automatic generation of automappings (automorphisms) from key constraints and value dependencies over XML schemas, and designing algebraic operations on mappings and schemas represented by automappings. During execution of mappings some missing or incomplete data may be inferred. A well-defined executable semantics for mappings and operations on mappings are proposed. A mapping language XDMap to specify XML schema mappings is proposed. The language allows to specify executable mappings that can be used to compute target instances from source instances preserving key constraints and value dependencies. The significance of mappings and operators over mappings is discussed on a scenario of data exchange in a P2P setting.

1 Introduction

Schema mapping is a basic problem for many applications such as data exchange, data integration, P2P databases or e-commerce, where data may be available at many different peers in many different schemas. [3, 6, 10, 11, 17, 20]. A schema mapping specifies a constraint that holds between schemas and can be thought of as a relation on instances. Executable mappings are mappings that are able to compute target instances from source instances preserving a set of given constraints [11].

Contributions. The main contributions of this work are as follows:

1. We propose a method for generating *automappings* over schemas from key and value dependency constraints defined in schemas. Automappings are then used to create mappings between schemas. Mappings can be combined (using the *Compose* and *Merge* operators) to give new mappings. We propose a mapping language, called XDMap, to mapping specification.
2. In the process of data transformation some missing or incomplete data, which are not given explicitly in sources, can be deduced based on value dependency constraints enforced by the target schema. We achieve that by representing missing data by terms defining the constraints. In some cases such terms may be resolved and replaced by the actual data.

Related work. We discuss our contribution against related work from the following three points of view.

1. A language for mapping specification. It is commonly accepted that the basic relationships between a source and a target relational schemas can be expressed as a source-to-target dependencies (STD) [2, 6, 11, 13]. In [3] $STDs$ are adopted to XML data in such a way that if a certain pattern occurs in the source, another pattern has to occur in the target. In our approach, the main idea of using $STDs$ consists in specifying how nodes in a target instance depend on key paths, how these key paths correspond to paths in sources, and how target values depend on other values. So, our approach is more operational and uses DOM interpretation of XML documents. To generate the instance of a target schema from instances of source schemas, we use the idea of *chasing* [2, 19]. In our mapping language XDMap we use Skolem functions with text-valued arguments from a source instance to create nodes (node identifiers) in a target instance. A concept of using Skolem functions for creation and manipulation of object identifiers has been previously proposed in ILOG [8] and in [1, 7]. Recently, Skolem functions are also used in some approaches to schema mappings, in Clio [16] are used for generating missing target values if the target element cannot be null (e.g. components of keys), in [19] are used in a query rewriting based on data mapping. Our mapping language can be compared with the mapping language proposed in [19]. However, XDMap is more powerful because we can use arbitrary Skolem functions for intermediate (non leaf-level) nodes, while in [19] these Skolem functions are system generated in a controlled way. Consequently, in our mappings, node generation is controlled by the mapping itself.

2. Generating mappings from key constraints. To define mappings we assume that key and some value constraints are specified within schema (using XML Schema [18] notation). We show how an automapping (a mapping from a schema onto itself) may be automatically generated from these constraints. It is significant in our approach that the constraints are specified outside the mapping by means of constraint-oriented notation. The generated automapping preserves these constraints. In contrast, in other mapping languages (e.g. in [19]) constraints must be explicitly encoded in the mapping language. This can make difficulties for future management when schemas evolve. To define correspondences between schema elements we use the method proposed in [9, 16] where a correspondence is defined between single elements from a source and a target schema. Establishing of a correspondence may be supported by automated techniques [17].

3. Algebra of mappings. Since a schema is represented by its automapping, we can operate over schemas and mappings in a uniform way. We discuss three operators over mappings (schemas): *Match* – creates a mapping between two schemas, *Compose* – combines two successive mappings, and *Merge* – produces a mapping that merges two source schemas. Operations on mappings, mainly composition, was recently studied in [6, 10–13]. Melnik *et al.* [11] have drawn our attention to *executable* mappings, i.e. mappings that compute target instances from source instances, and to the need of operations over such mappings.

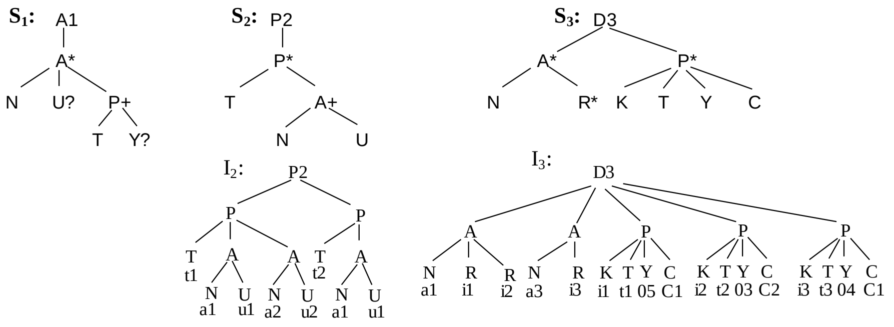
Outline. The paper is organized as follows. The next section shows a scenario of data exchange. Section 3 illustrates the problem of inferring some missing data in data exchange. Section 4 describes basic ideas of our approach to XML schema mapping and proposes syntax and semantics for the mapping language XDMap. Operations on mappings are discussed in Section 5. Section 6 concludes the paper.

2 A scenario of data exchange

We illustrate XML data exchange on a scenario of a P2P data exchanging system. Suppose there are three peers with schemas $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}_3$, respectively (Fig. 1). Only $\mathbf{S}_2$ and $\mathbf{S}_3$ are associated with data, while $\mathbf{S}_1$ is a mediated (or target) schema that does not store any data. The meaning of labels are: author (A), name (N) and university (U) of the author; paper (P) title (T), year (Y) of publication and the conference (C) where the paper has been presented. Elements labeled with R and K are used to join authors with their papers. I_2 and I_3 are instances of $\mathbf{S}_2$ and $\mathbf{S}_3$, respectively.

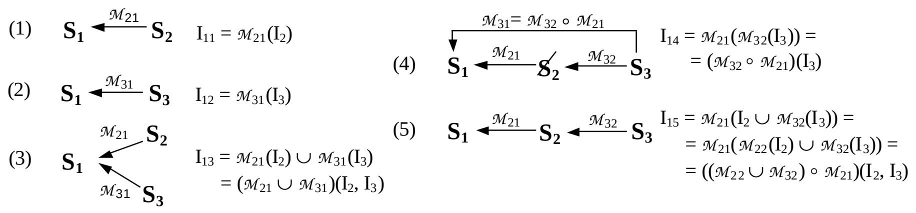
In such scenario we meet the problem of data exchange, i.e. computing target instances from source instances [3, 5, 11, 16, 19]. It is commonly agreed that mappings are needed to perform these functions effectively, where a mapping specifies a relationship between a set of source schemas and a target schema.

An instance of $\mathbf{S}_1$ can be obtained in different ways (Fig. 2): (1) and (2) by simple transformation of instance I_2 or I_3 by means of mappings $\mathscr{M}_{21}$ or $\mathscr{M}_{31}$; (3) as a merge $\mathscr{M}_{21} \cup \mathscr{M}_{31}$ over instances; (4) by means of composition $(\mathscr{M}_{32} \circ \mathscr{M}_{21})(\mathrm{I}_3)$ (when the $\mathbf{S}_2$'s peer is unavailable); (5) using combination of merge and composition, $((\mathscr{M}_{22} \cup \mathscr{M}_{32}) \circ \mathscr{M}_{21})(\mathrm{I}_2, \mathrm{I}_3)$. This shows that we often need to create new mappings from existing ones using composition and merging of mappings [6, 10, 11].



S1: A1
A*
N U? P+
T Y?
S2: P2
P*
T A+
N U
S3: D3
A* P*
N R* K T Y C
I2: P2
P P
T t1 A A T t2 A
N a1 U u1 N a2 U u2 N a1 U u1
I3: D3
A A P P P
N a1 R i1 R i2 N a3 R i3 K i1 T t1 Y 05 C C1 K i2 T t2 Y 03 C C2 K i3 T t3 Y 04 C C1

Fig. 1. Schemas: $\mathbf{S}_1, \mathbf{S}_2, \mathbf{S}_3$, and schema instances I_2 and I_3 ($\mathbf{S}_1$ does not have any stored instance)



(1) S1 ← M21 — S2 I11 = M21(I2)
(2) S1 ← M31 — S3 I12 = M31(I3)
(3) S1 ← M21 S2, S1 ← M31 S3 I13 = M21(I2) ∪ M31(I3) = (M21 ∪ M31)(I2, I3)
(4) M31 = M32 ∘ M21; S1 ← M21 — S2 ← M32 — S3 I14 = M21(M32(I3)) = = (M32 ∘ M21)(I3)
(5) S1 ← M21 — S2 ← M32 — S3 I15 = M21(I2 ∪ M32(I3)) = = M21(M22(I2) ∪ M32(I3)) = = ((M22 ∪ M32) ∘ M21)(I2, I3)

Fig. 2. Scenarios of data exchange – an instance of $\mathbf{S}_1$ may be computed in many ways

3 Using constraints for mapping specification

In our approach, we use two kinds of constraints to define mappings, namely:

1. *Value dependency constraints* (on the target) imposing that a value of a path depends on a tuple of values of other paths. We will declare them in the (non-standard) `<xs:valdep>` section of XML Schema (Fig. 4).
2. *Key constraints* (on a source) stating that a subtree is uniquely identified by a tuple of values of key paths [4, 18]. They are specified within `<xs:key>` and `<xs:keyref>` sections of XML Schema (Fig. 4).

Value dependencies can be used to infer missing data [3, 19]. Suppose we want to transform the instance I_2 under the target schema $\mathbf{S}_1$, i.e. an instance $\mathrm{I}_{11} = \mathscr{M}_{21}(\mathrm{I}_2)$ must be produced (Fig. 3(a)). The original instance provides no data about publication year. We know, however, that the publication year (Y) uniquely depends on the title (T) of the paper that is denoted by the constraint $Y = y(T)$, where y is the name of a function mapping titles into publication years. First, to Y we assign the term $y(t)$, where t is the title, as Y's text value. This convention forces some elements of type Y to have the same values (Fig. 3(a)). Such constraints are defined in the `<xs:valdep>` section in XML Schema (Fig. 4). Next, a term like $y(t)$ may be resolved using other mappings.

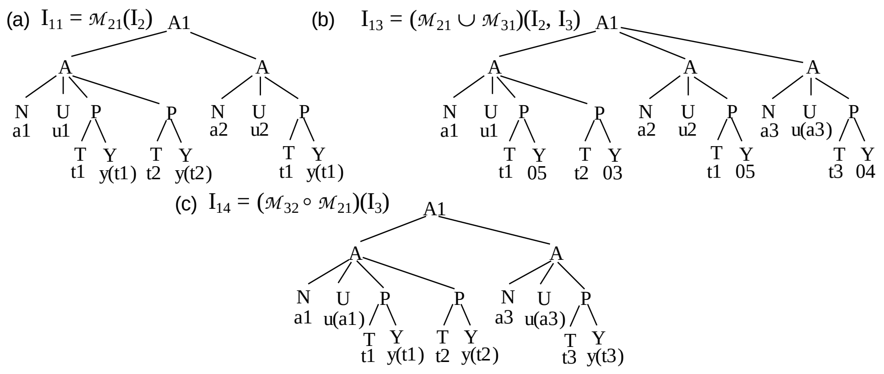

(a) $\mathrm{I}_{11} = \mathcal{M}_{21}(\mathrm{I}_2)$
A1
A
A
N a1
U u1
P
P
N a2
U u2
P
T t1
Y y(t1)
T t2
Y y(t2)
T t1
Y y(t1)
(b) $\mathrm{I}_{13} = (\mathcal{M}_{21} \cup \mathcal{M}_{31})(\mathrm{I}_2, \mathrm{I}_3)$
A1
N a1
U u1
N a2
U u2
N a3
U u(a3)
T t1
Y 05
T t2
Y 03
T t1
Y 05
T t3
Y 04
(c) $\mathrm{I}_{14} = (\mathcal{M}_{32} \circ \mathcal{M}_{21})(\mathrm{I}_3)$
A1
N a1
U u(a1)
N a3
U u(a3)
T t1
Y y(t1)
T t2
Y y(t2)
T t3
Y y(t3)

Fig. 3. Instances of schema $\mathbf{S}_1$ produced by mappings using constraints

Suppose we want to merge under $\mathbf{S}_1$ the instances I_{11} (Fig. 3(a)) and I_3 (Fig. 1). In this process terms denoting years will be replaced with actual values (Fig. 3(b)). Note that in this way we are able to infer the publication year of the paper written by $a2$. This information is not given explicitly neither in I_2 nor in I_3. The instances in Fig. 3(a)-(c) illustrate execution of composed mappings. We will address this issue deeply in Section 5.

Information provided by key constraints will be used to specify how many instances (nodes) of an element type must be in the computed target instance. For example, the element type $/A1/A$ in $\mathbf{S}_1$ is uniquely identified by the key path N. So, there are as many nodes of type $/A1/A$ as there are different values of $/A1/A/N$. In $\mathbf{S}_2$, however, elements of type A are identified by N but only in a context determined by the element type $/P2/P$ that is identified by T. Thus, to identify $/P2/P/A$ we need a pair of values determined by paths $/P2/P/T$ and $/P2/P/A/N$. In XML Schema such keys are defined using `<xs:key>`, and a declaration within a subelement denotes that the key identification is satisfied only in the context of superelement. In Fig. 4 there is a definition of the schema $\mathbf{S}_1$ written in an extended variant of XML Schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="A1">
  <xs:complexType><xs:sequence><xs:element ref="A"/></xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="A">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="N" type="xs:string"/>
    <xs:element name="U" type="xs:string" minOccurs="0"/>
    <xs:element ref="P" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  </xs:complexType>
  <xs:key name="AKey"><xs:selector xpath="."/><xs:field xpath="N"/>
  </xs:key>
  <xs:valdep>
   <xs:target name="U"/><xs:function name="u"/><xs:source xpath="N"/>
  </xs:valdep>
 </xs:element>
 <xs:element name="P">
  <xs:complexType>
  <xs:sequence>
    <xs:element name="T" type="xs:string"/>
    <xs:element name="Y" type="xs:string" minOccurs="0"/>
  </xs:sequence>
  </xs:complexType>
  <xs:key name="PKey"><xs:selector xpath="."/><xs:field xpath="T"/>
  </xs:key>
  <xs:valdep>
   <xs:target name="Y"/><xs:function name="y"/><xs:source xpath="T"/>
  </xs:valdep>
 </xs:element>
</xs:schema>
```

Fig. 4. XML Schema of $\mathbf{S}_1$, extended with `<xs:valdep>` declaration

4 Executable XML schema mappings

4.1 Basic ideas of mappings

From the definition in Fig. 4 we can generate the automapping $\mathscr{M}_{11}$ over $\mathbf{S}_1$ (Fig. 5), i.e. a mapping from $\mathbf{S}_1$ onto itself (numbers of lines provided here are for explanation only). It is formalized in Definition 2.

```
M11 = foreach G11 where Φ11 when C11 exists Δ11 =
      (1) foreach $y_{A1} in /A1, $y_A in $y_{A1}/A, $y_N in $y_A/N,
          $y_U in $y_A/U, $y_P in $y_A/P, $y_T in $y_P/T, $y_Y in $y_P/Y,
      (2) where true
      (3) when $y_U = u($y_N), $y_Y = y($y_T)
          exists
      (4) F_{/A1}() in F_{()}()/A1
      (5) F_{/A1/A}($y_N) in F_{/A1}()/A
      (6) F_{/A1/A/N}($y_N) in F_{/A1/A}($y_N)/N with $y_N
      (7) F_{/A1/A/U}($y_N,$y_U) in F_{/A1/A}($y_N)/U with $y_U
      (8) F_{/A1/A/P}($y_N,$y_T) in F_{/A1/A}($y_N)/P
      (9) F_{/A1/A/P/T}($y_N,$y_T) in F_{/A1/A/P}($y_N,$y_T)/T with $y_T
      (10) F_{/A1/A/P/Y}($y_N,$y_T,$y_Y) in F_{/A1/A/P}($y_N,$y_T)/Y with $y_Y
```

Fig. 5. Automapping $\mathscr{M}_{11}$ over $\mathbf{S}_1$

The clause **foreach** defines variables. Definitions in lines (1) and (2) are obvious. Equalities in (3) reflect value dependency constraints specified in the schema. They are interpreted as follows: While evaluating a constraint of the form $\$y = f(\overline{\$x})$, where $\overline{a}$ is the current value of the vector $\overline{\$x}$ of variables, the following actions are performed (see also Example 3 in Section 5):

1. If it is the first evaluation of the constraint for the argument $\overline{a}$, then the value of the term $f(\overline{a})$ is the textual representation "$f(\overline{a})$" of the term itself.
2. If $\$y$ is not bound to any value, then $\$y$ will be bound to the value of $f(\overline{a})$.
3. If $\$y$ is bound to b then b is assigned to $f(\overline{a})$ as its current value.

(4) creates two new nodes, the root r and the node n of the outermost element of type $/A1$, as results of Skolem functions $F_{()}()$ and $F_{/A1}()$, respectively. The node n is a child of type $A1$ of r. (5) creates a new node n' for any distinct value of $\$y_N$, each such node has the type $/A1/A$ and is a child of type A of the node created by $F_{/A1}()$ in (4). (6) For any distinct value of $\$y_N$ a new node n'' of type $/A1/A/N$ is created. Each such node is a child of type N of the node created by invocation of $F_{/A1/A}(\$y_N)$ in (5) for the same value of $\$y_N$. Because n'' is a leaf, it obtains the text value equal to the current value of $\$y_N$. Analogously for the rest of the specification.

4.2 Capturing key constraints by automappings

In a specification of automapping, Skolem functions and their arguments play a crucial role. We assume that:

– for any (rooted) path P in the schema there is exactly one Skolem function, $F_P(...)$, where F_P is the name of the Skolem function,
– arguments of a Skolem function $F_P(...)$ are determined by key paths defined for the element of type P in the schema.

In $\mathbf{S}_1$ there is exactly one root and one outermost element, so the corresponding Skolem functions have empty lists of arguments. Element of type $/A1/A$ has a key path N. Each its subelement inherits this key path and additionally has its local key paths. The local key paths for non-leaf elements are defined in the schema (Fig. 4). The local key path for a leaf element is, by default, this leaf element itself. Thus, for $\mathbf{S}_1$ we have the following key paths: N for $/A1/A$ and $/A1/A/N$; (N,T) for $/A1/A/P$ and $/A1/A/P/T$; and (N,T,Y) for $/A1/A/P/Y$. Text values of these key paths are bound to variables and are used as arguments of Skolem functions.

```
<xs:element name="A"><xs:complexType>...</xs:complexType>...
  <xs:keyref name="AKeyref" refer="PKey">
   <xs:selector xpath="."/>
   <xs:field xpath="R"/>
  </xs:keyref>
 </xs:element>
 <xs:element name="P"><xs:complexType>...</xs:complexType>
  <xs:key name="PKey">
   <xs:selector xpath="."/><xs:field xpath="K"/>
  </xs:key>
  <xs:valdep>
   <xs:target name="K"/><xs:function name="k"/>
   <xs:source xpath="T"/><xs:source xpath="N" ref="AKeyref"/>
  </xs:valdep> ...</xs:element>
```

Fig. 6. Fragment of XML Schema defining $\mathbf{S}_3$

In definition of $\mathbf{S}_3$ (Fig. 6), the schema specifies the *key* and *keyref* relationships between the K child element of the P element (the *referenced key*) and the R child element of the A element (the *foreign key*). Additionaly, the value dependency $K = k(T,N)$ says that the path N must start at element A referencing P via its foreign key defined in *AKeyref*. In the mapping specification, key references are captured as follows:

– in the **exists** clause any occurrence of a variable $\$z_R$ ranging over values of a foreign key is replaced with a variable $\$z_K$ ranging over values of the corresponding referenced key;
– the equality $\$z_R = \$z_K$ is inserted into the **where** clause.

Using these rules, we obtain the following specification of the automapping over $\mathbf{S}_3$ (we assume that the P element, with different value of K, is repeated for any co-author of the publication T):

$$
\begin{aligned}
\mathscr{M}_{33} = \ & \textbf{foreach } \$z_{D3} \ \underline{\text{in}} \ /D3,\ \$z_A \ \underline{\text{in}} \ \$z_{D3}/A,\ \$z_N \ \underline{\text{in}} \ \$z_A/N, \$z_R \ \underline{\text{in}} \ \$z_A/R,\\
& \qquad\qquad \$z_P \ \underline{\text{in}} \ \$z_{D3}/P,\ \$z_K \ \underline{\text{in}} \ \$z_P/K,\ \$z_T \ \underline{\text{in}} \ \$z_P/T,\\
& \qquad\qquad \$z_Y \ \underline{\text{in}} \ \$z_P/Y,\ \$z_C \ \underline{\text{in}} \ \$z_P/C\\
& \textbf{where } \$z_R = \$z_K\\
& \textbf{when } \$z_K = k(\$z_N, \$z_T), \$z_Y = y(\$z_T), \$z_C = c(\$z_T)\\
& \textbf{exists}\\
& F_{/D3}() \textbf{ in } F_{()}()/D3\\
& F_{/D3/A}(\$z_N) \textbf{ in } F_{/D3}()/A\\
& F_{/D3/A/N}(\$z_N) \textbf{ in } F_{/D3/A}(\$z_N)/N \textbf{ with } \$z_N\\
& F_{/D3/A/R}(\$z_N, \$z_K) \textbf{ in } F_{/D3/A}(\$z_N)/R \textbf{ with } \$z_K\\
& F_{/D3/P}(\$z_K) \textbf{ in } F_{/D3}()/P\\
& F_{/D3/P/K}(\$z_K) \textbf{ in } F_{/D3/P}(\$z_K)/K \textbf{ with } \$z_K\\
& F_{/D3/P/T}(\$z_K, \$z_T) \textbf{ in } F_{/D3/P}(\$z_K)/T \textbf{ with } \$z_T\\
& F_{/D3/P/Y}(\$z_K, \$z_Y) \textbf{ in } F_{/D3/P}(\$z_K)/Y \textbf{ with } \$z_Y\\
& F_{/D3/P/C}(\$z_K, \$z_C) \textbf{ in } F_{/D3/P}(\$z_K)/C \textbf{ with } \$z_C
\end{aligned}
$$

4.3 Syntax and semantics for mappings

In general, there are two vectors of variables $\overline{\$x}$ and $\overline{\$y}$ in a mapping $\mathscr{M}$. Variables from $\overline{\$x}$ are bound in a source by means of the **foreach** clause, and variables from $\overline{\$y}$ are bound to terms in the **when** clause as a consequence of a value dependency constraints. The part **foreach/where/when** of a mapping $\mathscr{M}(\overline{\$x}; \overline{\$y})$ determines a partially ordered set $(\Omega, \leq)$ of bindings of $\mathscr{M}$'s variables. For example, in the mapping $\mathscr{M}_{21}$ (Fig. 7) for two bindings $\omega_1, \omega_2 \in \Omega$ over I$_2$, where $\omega_1 = (\$x_T \to t_1, \$x_N \to a_1, \$x_U \to u_1, \$y_Y \to y(t_1))$ and $\omega_2 = (\$x_T \to t_1, \$x_N \to a_2, \$x_U \to u_2, \$y_Y \to y(t_2))$, we have $\omega_1 < \omega_2$, because the tuple of leaf nodes providing values for ω_1 precedes the tuple of leaf nodes providing values for ω_2. Bindings from Ω are used in the **exists** part to produce the result target instance. The ordering imposed in Ω by a source instance should be preserved in the target instance.

If the **foreach/where** clause is defined over $\mathbf{S}_2$, while the **when/exists** concerns $\mathbf{S}_1$, then we deal with a mapping $\mathscr{M}_{21}$ from $\mathbf{S}_2$ into $\mathbf{S}_1$. Then, after the given replacement of variables, we obtain:

$$
\begin{aligned}
\mathscr{M}_{21} = \ & \textbf{foreach } \$x_{P2} \ \underline{\text{in}} \ /P2,\ \$x_P \ \underline{\text{in}} \ \$x_{P2}/P,\ \$x_T \ \underline{\text{in}} \ \$x_P/T,\\
& \qquad\quad \$x_A \ \underline{\text{in}} \ \$x_P/A,\ \$x_N \ \underline{\text{in}} \ \$x_A/N,\ \$x_U \ \underline{\text{in}} \ \$x_A/U\\
& \textbf{where true}\\
& \textbf{when } C_{11}(\$y_N, \$y_U, \$y_T, \$y_Y)[\$y_N \to \$x_N, \$y_U \to \$x_U, \$y_T \to \$x_T]\\
& \textbf{exists } \Delta_{11}(\$y_N, \$y_U, \$y_T, \$y_Y)[\$y_N \to \$x_N, \$y_U \to \$x_U, \$y_T \to \$x_T]
\end{aligned}
$$

Fig. 7. Mapping $\mathscr{M}_{21}$ from $\mathbf{S}_2$ into $\mathbf{S}_1$

Thus, the **when** clause of $\mathscr{M}_{21}$ is equal to $\$x_U = u(\$x_N), \$y_Y = y(\$x_T)$. There is no replacement for $\$y_Y$, so its value must be set as the current value of the term $y(\$x_T)$, according to the rules (1)–(3) discussed in p. 4.1. We set it as the term $y(t)$, where t is the current value of $\$x_T$ (see Fig. 3(a)). It is a form of *Skolemization*. Thus, a mapping specification in XDMap conforms to the general form of *source-to-target generating dependencies* [2, 6, 13]:

$$\forall \overline{\$x}(G(\overline{\$x}) \wedge \Phi(\overline{\$x}) \Rightarrow \exists \overline{\$y} C(\overline{\$x}; \overline{\$y}) \wedge \Delta(\overline{\$x}; \overline{\$y})),$$

where $G(\overline{\$x})$ and $\Phi(\overline{\$x})$ are conjunctions of atomic formulas over a source, and $C(\overline{\$x};\overline{\$y})$ and $\Delta(\overline{\$x};\overline{\$y})$ are conjunctions of atomic formulas over a target.

Definition 1. *An executable schema mapping in XDMap (or mapping for short) between a source schema* **S** *and a target schema* **T** *is a sequence* $\mathscr{M} ::= (M,...,M)$ *of mapping rules between* **S** *and* **T***, where:*

$$
\begin{aligned}
M = (G,\Phi,C,\Delta)(\overline{\$x};\overline{\$y}) := \; & \textbf{foreach } G(\overline{\$x}) \\
& \textbf{where } \Phi(\overline{\$x}) \\
& \textbf{when } C(\overline{\$x};\overline{\$y}) \\
& \textbf{exists } F_{P/l}(\overline{\$x};\overline{\$y}) \textbf{ in } F_P(\overline{\$x'};\overline{\$y'})/l[\textbf{ with } \$x''\ ]
\end{aligned}
$$

- *G is a list of variable definitions over a source schema;*
- *Φ is a conjunction of atomic conditions: $\$x = \$x'$ or $\$x \neq \$x'$;*
- *C is a list of target constraints: $\$x = f(\overline{\$x})$ or $\$y = f(\overline{\$x})$, $\$x \in \overline{\$x}$, $\$y \in \overline{\$y}$;*
- *$F_P(\overline{\$x};\overline{\$y})$ is a Skolem term, where P is a rooted path in a target schema;*
- *$(\overline{\$x'};\overline{\$y'}) \subseteq (\overline{\$x};\overline{\$y})$, $\$x'' \in (\overline{\$x};\overline{\$y})$.* □

Semantics for XDMap is defined as follows:

Definition 2. *Let $\mathscr{M} = (G,\Phi,C,\Delta)(\overline{\$x};\overline{\$y})$ be a mapping, and $(\Omega,\leq)$ be a partially ordered set of bindings of variables $(\overline{\$x};\overline{\$y})$ determined by (G,Φ,C). A target instance* I *of a target schema* **T** *is then obtained as follows:*

1. *$F_{()}()$ – the root of* I.
2. *$F_P(\overline{\$x};\overline{\$y})(\omega) = n$ – a node of type P.*
3. *If $F_{P/l}(\overline{\$x};\overline{\$y})(\omega) = n$, $F_P(\overline{\$x'};\overline{\$y'})(\omega) = n'$ and $(\overline{\$x'};\overline{\$y'}) \subseteq (\overline{\$x};\overline{\$y})$, then n is a child of type l of the node n'.*
4. *Let $F_{P/l}(\overline{\$x};\overline{\$y})(\omega_1) = n_1$, $F_{P/l}(\overline{\$x};\overline{\$y})(\omega_2) = n_2$, $\omega_1 \leq \omega_2$, and $(\overline{\$x'};\overline{\$y'})(\omega_1) = (\overline{\$x'};\overline{\$y'})(\omega_2)$, then $n_1 \leq n_2$ in the document order in the set of children of type l of the node $F_P(\overline{\$x'};\overline{\$y'})(\omega_1)$.*
5. *If $F_{P/l}(\overline{\$x};\overline{\$y})(\omega) = n$ is a leaf, then the text value of n is $\omega(\$x'')$.*

5 Operations on mappings

Mappings can be combined by means of some operators giving a result that in turn is a mapping. We define the following operations: *Match*, *Compose*, and *Merge*. First, we have to define a *correspondence* between *maximal paths*, i.e. paths leading from a root to a leaf in a schema. Establishing the correspondence is a crucial task in definition of data mappings [17].

Definition 3. *Let $\mathscr{P}$ and $\mathscr{P}'$ be sets of maximal paths from schemas* **S** *and* **S**$'$*, respectively. A correspondence from* **S** *into* **S**$'$ *is a partial injection $\sigma : \mathscr{P} \to \mathscr{P}'$ which maps a path $P \in \mathscr{P}$ on a path $P' = \sigma(P) \in \mathscr{P}'$.* □

Example 1. A correspondence between $\mathbf{S}_1$ and $\mathbf{S}_2$ (Fig. 1) is: $\sigma(/A1/A/N) = /P2/P/A/N$, $\sigma(/A1/A/U) = /P2/P/A/U$, $\sigma(/A1/A/P/T) = /P2/P/T$. □

Match and Compose

Let $\mathscr{M}_s$ and $\mathscr{M}_t$ be two automappings over **S** and **T**, respectively. Then $Match(\mathbf{S},\mathbf{T}) = \mathscr{M}_s \circ \mathscr{M}_t$ is a mapping from **S** into **T**.

Let $\mathscr{M}_1$ and $\mathscr{M}_2$ be mappings from $\mathbf{S}_1$ into $\mathbf{S}_2$ and from $\mathbf{S}_2$ into $\mathbf{S}_3$, respectively. Then $Compose(\mathscr{M}_1,\mathscr{M}_2) = \mathscr{M}_1 \circ \mathscr{M}_2$ is a mapping from $\mathbf{S}_1$ to $\mathbf{S}_3$.

The composition $\mathscr{M} = \mathscr{M}_1 \circ \mathscr{M}_2$, where $\mathscr{M}_1 = (G_1,\Phi_1,C_1,\Delta_1)(\overline{\$x_1};\overline{\$y_1})$, $\mathscr{M}_2 = (G_2,\Phi_2,C_2,\Delta_2)(\overline{\$x_2};\overline{\$y_2})$ and σ is a correspondence from $source(\mathscr{M}_2)$ into $source(\mathscr{M}_1)$, is a mapping $\mathscr{M} = (G,\Phi,C,\Delta)(\overline{\$x};\overline{\$y})$ defined by the following replacement ($\$v : P$ denotes that the variable $\$v$ is of type P, i.e. ranges over text values of P):

$$\mathscr{M} = (G_1,\Phi_1,C_2,\Delta_2)[\$v : P \to \$w : \sigma(P) \mid \$v \in \overline{\$x_2}, \$w \in \overline{\$x_1}].$$

The result of the replacement is defined as follows:

- any occurrence of a variable $\$v$ of type P in $(G_1,\Phi_1,C_2,\Delta_2)$, $\$v \in \overline{\$x_2}$, is replaced by a variable $\$w \in \overline{\$x_1}$ of type $\sigma(P)$, $\$w$ is a *replacing variable*;
- $\overline{\$x}$ consists of all replacing variables and all variables occurring in Φ_1;
- $\overline{\$y}$ consists of all variables occurring in C_2 which have not been replaced;
- (G,Φ,C,Δ) is created from $(G_1,\Phi_1,C_2,\Delta_2)$, where all replacements have been done and, additionally, from G_1 all unnecessary variable definitions have been removed.

Example 2. The mapping $\mathscr{M}_{21}$ (Fig. 7) is a composition of $\mathscr{M}_{22}$ and $\mathscr{M}_{11}$, i.e.
$\mathscr{M}_{21}(\$x_N,\$x_U,\$x_T;\$y_Y) = \mathscr{M}_{22}(\$x_T,\$x_N,\$x_U) \circ \mathscr{M}_{11}(\$y_N,\$y_U,\$y_T,\$y_Y)$.
Similarly we can show that:
$\mathscr{M}_{32}(\$z_T,\$z_N,\$z_R,\$z_K;\$v_U) =$
$= \mathscr{M}_{33}(\$z_N,\$z_R,\$z_K,\$z_T,\$z_Y,\$z_C) \circ \mathscr{M}_{22}(\$v_T,\$v_N,\$v_U)$.
$\mathscr{M}_{321}(\$z_T,\$z_N,\$z_R,\$z_K;\$x_U,\$y_Y) = \mathscr{M}_{32}(\$z_T,\$z_N,\$z_R,\$z_K;\$v_U)\circ$
$\circ\mathscr{M}_{21}(\$x_N,\$x_U,\$x_T;\$y_Y) =$
$=$ **foreach** $G_{32}(\$z_N,\$z_R,\$z_K,\$z_T)$
where $\$z_R = \$z_K$
when $\$x_U = u(\$z_N), \$y_Y = y(\$z_T)$ $(= C_{21}[\$x_N \to \$z_N, \$x_T \to \$z_T])$
exists $\Delta_{21}(\$x_N,\$x_U,\$x_T;\$y_Y)[\$x_N \to \$z_N, \$x_T \to \$z_T]$
where
$\Delta_{321} = F_{/A1}()$ **in** $F_{()}()/A1$
$F_{/A1/A}(\$z_N)$ **in** $F_{/A1}()/A$
$F_{/A1/A/N}(\$z_N)$ **in** $F_{/A1/A}(\$z_N)/N$ **with** $\$z_N$
$F_{/A1/A/U}(\$z_N,\$x_U)$ **in** $F_{/A1/A}(\$z_N)/U$ **with** $\$x_U$
$F_{/A1/A/P}(\$z_N,\$z_T)$ **in** $F_{/A1/A}(\$z_N)/P$
$F_{/A1/A/P/T}(\$z_N,\$z_T)$ **in** $F_{/A1/A/P}(\$z_N,\$z_T)/T$ **with** $\$z_T$
$F_{/A1/A/P/Y}(\$z_N,\$z_T,\$y_Y)$ **in** $F_{/A1/A/P}(\$z_N,\$z_T)/Y$ **with** $\$y_Y$

$\mathscr{M}_{321}$ has two variables, $\$x_U$ and $\$y_Y$, which are not bound in the source. Instead, they are bound in the **when** clause to target terms $u(\$z_N)$ and $y(\$z_T)$, respectively. An instance of the mapping is given in Fig. 3(c). All term-valued leaves may be either removed, replaced with nulls, or left as they are (they may be resolved and replaced with actual values in next mappings as in Fig. 3(a)-(b)).

Merge

Let $\mathscr{M}_1$ and $\mathscr{M}_2$ be mappings from $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively, into $\mathbf{S}_3$. Then a mapping that *merges* $\mathbf{S}_1$ and $\mathbf{S}_2$ under $\mathbf{S}_3$ is the mapping $Merge(\mathscr{M}_1, \mathscr{M}_2) = \mathscr{M}_1 \cup \mathscr{M}_2$ defined as follows. Let $\mathscr{M}_1 = (G_1, \Phi_1, C_1, \Delta_1)(\overline{\$x_1}; \overline{\$y_1})$ and $\mathscr{M}_2 = (G_2, \Phi_2, C_2, \Delta_2)(\overline{\$x_2}; \overline{\$y_2})$, where $(\overline{\$x_1}; \overline{\$y_1})$ and $(\overline{\$x_2}; \overline{\$y_2})$ are disjoint. Then:

$$\mathscr{M}_1 \cup \mathscr{M}_2 = (G_1 \cup G_2, \Phi_1 \wedge \Phi_2, C_1 \wedge C_2, \Delta_1 \cup \Delta_2)(\overline{\$x_1}; \overline{\$x_2}; \overline{\$y_1}; \overline{\$y_2}).$$

Example 3. The merge $\mathscr{M}_4 = \mathscr{M}_{21} \cup \mathscr{M}_{31}$ is understood as a mapping consisting of all mapping rules from $\mathscr{M}_{21}$ and those from $\mathscr{M}_{31}$. Below, we show only these rules that involve variables from constraints in the **when** clause only.

$\mathscr{M}_4 =$ **foreach** $G_{22}(\$x_T, \$x_N, \$x_U), G_{33}(\$z_N, \$z_R, \$z_K, \$z_T, \$z_Y, \$z_C)$
where $\$z_R = \$z_K$
when $\$x_U = u(\$x_N), \$y_Y = y(\$x_T), \$y_U = u(\$z_N), \$z_Y = y(\$z_T)$
exists ...
(1) $F_{/A1/A/U}(\$x_N, \$x_U)$ **in** $F_{/A1/A}(\$x_N)/U$ **with** $\$x_U$
(2) $F_{/A1/A/P/Y}(\$x_N, \$x_T, \$y_Y)$ **in** $F_{/A1/A/P}(\$x_N, \$x_T)/Y$ **with** $\$y_Y$
(3) $F_{/A1/A/U}(\$z_N, \$y_U)$ **in** $F_{/A1/A}(\$z_N)/U$ **with** $\$y_U$
(4) $F_{/A1/A/P/Y}(\$z_N, \$z_T, \$z_Y)$ **in** $F_{/A1/A/P}(\$z_N, \$z_T)/Y$ **with** $\$z_Y$
...

The current values of terms from the **when** clause at different stages of processing of the mapping $\mathscr{M}_4$, are shown below (see rules (1)-(4) in p. 4.1). The current value of the term is bound to a variable in the **when** clause. This binding is enforced by constraints defined in $\mathbf{S}_1$.

$Step$	$u(a1)$	$u(a2)$	$u(a3)$	$y(t1)$	$y(t2)$	$y(t3)$
(1)	$u1$	$u2$	–	–	–	–
(2)	$u1$	$u2$	–	”$y(t1)$”	”$y(t2)$”	–
(3)	$u1$	$u2$	”$u(a3)$”	”$y(t1)$”	”$y(t2)$”	–
(4)	$u1$	$u2$	”$u(a3)$”	05	03	04

Execution of this mapping is illustrated in Fig. 3. Fig. 3(a) shows the result produced by the part corresponding to $\mathscr{M}_{21}$ (after step (2)), and Fig. 3(b) the final result (after step (4). Note, that the term ”$u(a_3)$” cannot be resolved. □

6 Conclusion

We have described a novel approach to XML schema mapping specification and operations over schema mappings. We discussed how automappings may be generated using key constraints [4, 18], keyref constraints [18], and some value dependency constraints defined in XML Schema. Constraints on values can be used to infer some missing data. Mappings between two schemas can be generated automatically from their automappings and correspondences between maximal paths of these two schemas. Automappings represent schemas, so operations over schemas and mappings can be defined and performed in a uniform way. We propose some algebraic operations over mappings. The syntax and semantics for the mapping language XDMap are defined and discussed. Our techniques can be applied in various XML data exchange scenarios, and are especially useful when the set of data sources change dynamically (e.g. in P2P environment) [14, 15].

References

1. Abiteboul, S., Buneman, P., Suciu, D.: *Data on the Web. From Relational to Semistructured Data and XML*, Morgan Kaufmann, San Francisco, 2000.
2. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*, Addison-Wesley, Reading, Massachusetts, 1995.
3. Arenas, M., Libkin, L.: XML Data Exchange: Consistency and Query Answering, *PODS Conference*, 2005, 13–24.
4. Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., Tan, W. C.: Reasoning about keys for XML, *Information Systems*, **28**(8), 2003, 1037–1063.
5. Fagin, R., Kolaitis, P. G., Popa, L.: Data exchange: getting to the core., *ACM Trans. Database Syst.*, **30**(1), 2005, 174–210.
6. Fagin, R., Kolaitis, P. G., Popa, L., Tan, W. C.: Composing Schema Mappings: Second-Order Dependencies to the Rescue., *PODS*, 2004, 83–94.
7. Fernandez, M. F., Florescu, D., Kang, J., Levy, A. Y., Suciu, D.: Catching the Boat with Strudel: Experiences with a Web-Site Management System., *SIGMOD Conference*, 1998, 414–425.
8. Hull, R., Yoshikawa, M.: ILOG: Declarative Creation and Manipulation of Object Identifiers., *VLDB*, 1990, 455–468.
9. Kuikka, E., Leinonen, P., Penttonen, M.: Towards automating of document structure transformations., *ACM Symposium on Document Engineering*, 2002, 103–110.
10. Madhavan, J., Halevy, A. Y.: Composing Mappings Among Data Sources., *VLDB*, 2003, 572–583.
11. Melnik, S., Bernstein, P. A., Halevy, A. Y., Rahm, E.: Supporting Executable Mappings in Model Management., *SIGMOD Conference*, 2005, 167–178.
12. Melnik, S., Rahm, E., Bernstein, P. A.: Rondo: A Programming Platform for Generic Model Management., *SIGMOD Conference*, 2003, 193–204.
13. Nash, A., Bernstein, P. A., Melnik, S.: Composition of Mappings Given by Embedded Dependencies., *PODS*, 2005.
14. Pankowski, T.: Specifying Schema Mappings for Query Reformulation in Data Integration Systems, 3rd *Atlantic Web Intelligence Conference - AWIC'2005*, Lecture Notes in Computer Science 3528, Springer-Verlag, 2005, 361–365.
15. Pankowski, T.: Integration of XML Data in Peer-To-Peer E-commerce Applications, 5th *IFIP Conference I3E'2005*, Springer, New York, 2005, 481–496.
16. Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., Fagin, R.: Translating Web Data., *VLDB*, 2002, 598–609.
17. Rahm, E., Bernstein, P. A.: A survey of approaches to automatic schema matching, *The VLDB Journal*, **10**(4), 2001, 334–350.
18. XML Schema Part 1: Structures 2d Edition: 2004. www.w3.org/TR/xmlschema-1
19. Yu, C., Popa, L.: Constraint-Based XML Query Rewriting For Data Integration., *SIGMOD Conference*, 2004, 371–382.
20. Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings when Schemas Evolve., *VLDB*, 2005, 1006–1017.