Evaluating Nested Queries on XML Data

Carlo Sartiani
Dipartimento di Informatica
Università di Pisa
Via Buonarroti 2, Pisa, Italy
e-mail: sartiani@di.unipi.it

Abstract

In the past few years, much attention has been paid to the study of semistructured *data, i.e., data with irregular, possibly unstable, and rapidly changing structure, and, in particular, to the study of their best-known incarnation: XML. The growing interest toward XML leaded to the definition of many querying and manipulating tools, such as the standard query language XQuery [3].*

Unlike SQL, XQuery poses no restriction on query nesting (a XQuery query can be nested wherever a well-formed XML document is expected), and lacks explicit clauses for performing group-by *operations. As a result, nested queries play an important role in the context of XQuery.*

This paper shows the techniques used in the Xtasy system for processing and evaluating nested queries on XML data. These techniques applies to the physical level only, and they are based on the massive use of node sharing*, with the aim of decreasing as much as possible the use of secondary storage. The proposed solutions are general enough to be applied, with a few modifications, to more structured contexts.*

1. Introduction

In the past few years, much attention has been paid to the study of *semistructured* data, i.e., data with irregular, possibly unstable, and rapidly changing structure, and, in particular, to the study of their best-known incarnation: XML. XML is a universal data representation format that can be used to describe any kind of information, from highly structured data (e.g., relational databases) to loosely structured documents (e.g., bibliographic documents) as well as unstructured documents (e.g., texts); given its flexibility in data representation and its *auto-descriptiveness*, XML became the *de-facto* standard representation format for semistructured data.

While the prevalent use of XML is data interchange between applications, for the purpose of interoperability and/or integration [4] [2], XML allows one to naturally represent and manipulate particular kinds of data that would not fit well a highly structured context: genetic data, paleobiological data, movies [12], etc. The growing interest toward XML leaded to the definition of many querying and manipulating tools, and, in particular, to the standard query language XQuery [3].

XQuery is a functional, *strongly typed*, Turing-complete query language designed by W3C as an evolution of past object-oriented and semistructured query languages. Unlike SQL, XQuery poses no restriction on query nesting (a XQuery query can be nested wherever a well-formed XML document is expected), and, at the time we write this paper, it lacks explicit clauses for performing *group-by* operations. As a result, nested queries play an important role in the context of XQuery. Despite this importance, little attention has been devoted to the problem of efficiently processing and evaluating nested queries in XQuery. While techniques developed for relational and OO databases can be partially adapted to this context, only a small fraction of nested queries can be *unnested* at compile time; moreover, the ordered, tree-structured shape of XML data makes the processing of nested queries much more complex than in the relational case (as shown in [13], the same happens for recursive query optimization and evaluation), so there is the need for techniques for efficiently processing nested queries on XML data.

Our Contribution This paper shows the techniques used in Xtasy [1] for processing and evaluating nested queries on XML data. These techniques applies to the physical level only, hence they are used for *unnestable* queries only (logical rewriting rules for nested queries on XML data are described in [14]); they are based on the massive use of *node sharing*, with the aim of decreasing as much as possible the use of secondary storage, and of producing a good degree of scalability. The proposed solutions are general enough

to be applied, with a few modifications, to more structured contexts.

Paper Outline The paper is structured as follows. Sections 2 and 3 briefly describe the storage scheme of Xtasy, as well as its main physical operators. Section 4, then, illustrates the techniques adopted for processing nested queries. In Sections 5 and 6, finally, we comment on some related works, and draw our conclusions.

2. Storage scheme

Xtasy storage scheme was designed with the aim of supporting XML updates; hence, sophisticated *IR-like* data structures [7] were discarded in favor of simpler and more easily *updatable* data structures. The result is a storage scheme that allows the system to access data relatively fast, and to support good update rates.

Xtasy storage scheme is based on the clustering of elements and attributes according to their tags and names; node clustering is combined with the use of *parent/child* pointers, hence making this scheme a combination of relational and object-oriented storage schemes (see [17] for a taxonomy of storage schemes for XML data). Element and attribute clustering is combined with the use of an external index (the Structural Index) for storing the parent/child relationship. As a consequence, XML nodes can be accessed by scanning the correspondent cluster, or by looking up the index.

2.1. Node clustering and representation

Given a XML tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$, each node $n \in N$ is endowed with a pair of integer numbers $(pos, endpos)$, denoting respectively its position in a preorder visit and the position of its last descendant (the rightmost one); given a node n_e, $pos(n_e)$ may also be regarded as the position of the node in the tag (respectively, attribute name) opening partial order.

From the definition of pos and $endpos$, it follows that, given two nodes x and y, x is ancestor of y if and only if $pos(x) < pos(y)$ and $endpos(x) \geqslant endpos(y)$. This relation is very important for efficiently evaluating // operations, since it allows the system to quickly check for the ancestor/descendant relationship.

Given a XML tree $\mathcal{T}$, its nodes are distributed in heapfiles according to their *types*. Specifically, the system creates:

- one heapfile per tag name, e.g., the heapfile `*book*` for book elements;
- one heapfile per attribute name, e.g., the heapfile `**class**` for class attributes;
- one heapfile for storing values, e.g., the heapfile `***value***`.

The system associates a unique fileID to each heapfile (for obvious reasons); this, together with the encapsulation of tag and attribute names into heapfile names, allows the system to use integers in place of tag and attribute names, hence introducing a very modest form of data compression.

Element and attributes are represented as fixed-length record containing the following fields: a), the position and the end position of the node, and b), the pointer to the father node.

Node pointers are represented by EIDs (Element IDs), where the EID of a node n is a pair containing the fileID of the heapfile enclosing n and the record identifier of n; each XML node, hence, has its own unique EID, which can be used for directly accessing the node and for inspecting its *type* (element, attribute, or value): in particular, EIDs of element and attribute nodes denote also their tags or names, thus filtering a list of EIDs according to a given tag or name requires no persistent storage access.

Value nodes are, instead, represented as variable-length records containing: a), the position of the node (the end position of the node is equal to its position); b), the father EID; and c), the string representation of the value itself.

The choice of keeping values separate from their enclosing elements and attributes allows an easy representation of *mixed content* element.

2.2. The Structural Index

Even though the only use of back pointers is sufficient to reconstruct the structure of the whole tree, evaluating path expressions with the only aid of back pointers is a nightmare. For this purpose, Xtasy stores the parent/child relationship in the Structural Index, which is just a $B^+ - tree$ associating each element EID with the EIDs of its children nodes.

The use of a $B^+ - tree$ instead of more complex index structures has a threefold motivation. First of all, the $B^+ - tree$ is a simple and well-known data structure that can be easily managed; second, the $B^+ - tree$ allows a better handling of updates than IR-like data structures, and it can be easily adapted to support XML update languages [16]; finally, the $B^+ - tree$ allows a relatively easy dynamic indexing of synthetic elements created during nested query evaluation, which is a *must* in the case of free nesting languages as XQuery. These advantages are counterbalanced by a significant space overhead w.r.t. structures combining IR and compression techniques [7].

The main advantages of the storage scheme of Xtasy are the support for efficient execution of path operations such as //, the support for updates as well as the relatively

easy management of nested queries; these *pros*, however, are counterbalanced by a significant space overhead (mostly due to the Structural Index), and by the need of dynamically reconstructing XML fragments during the execution of $return$-like operations.

3. Physical operators overview

Xtasy physical operators are based on the iterative execution model [9], where each operator works on a single *data granule* per time, hence decreasing the memory requirements and enhancing the scalability of the system.

Physical operators communicate through method invocation, and exchange a particular kind of data granule called *unboxed tuple*. Even if its name is a true non-sense, the unboxed tuple is designed for combining tuple-based and slot-based data granules, therefore eliminating the need for two different data granule kinds: indeed, path evaluation operators work on document and indexes, hence manipulating EIDs, while other operators work on variable binding tuples.

An unboxed tuple is composed by a vector of pairs $(var\_name, EID\ sequence)$, and by a slot for a single free EID.

The vector part is used for hosting variable bindings collected during query evaluation, and it is accessed by most physical operators; the free EID slot, instead, is used for carrying EIDs collected during path evaluation. In particular, these EIDs are produced by path evaluation operators, and consumed by path evaluation and binding operators.

Xtasy operators can be roughly divided into two categories: *unary* operators, which have at most one input operator, and *binary* operators, which have two input operators. Unary operator class contains operators such as XMLCut (projection without duplicate elimination), XMLDistinct (duplicate elimination), MergeSort (sorting operator), XMLSelection (predicate evaluation), XMLBinder (variable binding), XMLReturn (construction of query result), and XMLSeqPath (path evaluation). Binary operator class, instead, contains XMLUnion (set union without duplicate elimination), XMLExternalUnion (external union), XMLNestedLoopJoin (general purpose join operator), and XMLDependentJoin (dependent join operator). The operators directly involved in nested query evaluation are XMLReturn and XMLDependentJoin, so it is worth to take a deeper look inside them.

3.1. XMLReturn

XMLReturn is used for creating new XML fragments by filling a XML skeleton with variable values. XMLReturn comes in two versions. XMLFinalReturn produces the final result of a query, which can be serialized on screen or on disk. Therefore, XMLFinalReturn fills skeleton gaps with the XML trees pointed by the EIDs extracted from tuples; these trees are reconstructed on the fly, which makes this operator quite expensive.

XMLNestedReturn, instead, produces the result of a nested query, and it is described in detail in Section 4.

3.2. XMLDependentJoin

This operator behaves as a standard join operator, with the only exception that tuples from the left input are passed to the right operand, and used for instantiating and evaluating the right input. Indeed, this operator can be used for performing d-joins, and it is introduced only during the compilation of nested queries.

4. Nested query evaluation

As previously noted, the system uses XMLNestedReturn for producing results of nested queries: this operator takes care of most of the issues regarding nested query processing.

Before describing our approach, and seeing how XMLNestedReturn works, it is necessary to briefly analyze what kind of results nested queries produce, as well as the main issues related to them. The following example shows a typical use of nested queries.

Example 4.1 Consider the bibliographic database previously illustrated and the following query, which returns, for each author, the list of her works.

```
for $a in input()/*/author
let $title_list :=
      for $b in input()/*,
          $t in $b/title
      where some $ba in $b/author
            satistfies ($ba is $a)
      return <pubtitle> data($t) </pubtitle>
return <ref> { $a, $title_list} </ref>
```

The inner query selects the titles of works published by any given author, and returns their values enclosed in the new tag pubtitle. As prescribed by XQuery Formal semantics [6], `data($t)` nodes have new object identifiers, and their positions should refer to the new enclosing context. ■

This sample query shows that nested queries may return newly created XML nodes (e.g., pubtitle elements), as well as nodes directly copied from the existing database (e.g., `data($t$)`). Hence, it is worth to classify XML nodes according to the way they are built; this taxonomy will greatly simplify the discussion.

Definition 4.2 *XML nodes (regardless of their type) can be classified as follows:*

- natural *nodes: nodes existing in the persistent database;*
- artificial *nodes: nodes existing in the persistent database and copied into the result of queries;*
- synthetic *nodes: newly created nodes by nested queries.*

Artificial nodes have different OIDs w.r.t. their natural versions, and assume new positional information (they are considered part and parcel of a new document). XMLNestedReturn, hence, have to generate synthetic and artificial nodes, to assign them new positional information, to refresh artificial node oids, as well as to generate new Structural Index fragments.

For the sake of simplicity, we will consider only element and attribute nodes, and we will not take into account reference semantics RETURN clauses.

4.1. Synthetic nodes and extended EIDs

For supporting synthetic nodes, the system must set them apart from natural nodes, since they do not survive the execution of the outer query. To this end, the use of simple EIDs, as shown in Section 2, is not sufficient, because they discriminate nodes by their name and not by their provenance; hence, we associate each query execution with a unique ID, which is then used for extending EIDs. QueryIDs uniquely identify query executions (the same nested query can be instantiated and invoked many times during outer query evaluation), and they are assigned both at compile time and at run time: during query parsing, the system creates a unique ID for each query block; at run time, then, each execution of the query generates a new dynamic ID (*invocation ID*). Hence, each query has associated a set of QueryIDs of the form $\{queryBlock.queryExec\}$.

QueryIDs are volatile, since they perish after outer query execution. The following example shows the assignment of QueryIDs.

Example 4.3 Consider the following query, returning book authors grouped with the titles of their books, and with their publisher.

```
for $a in input()/book/author
let $title_list :=
       for $b in input()/book,
           $t in $b/title
       where some $ba in $b/author
             satistfies ($ba is $a)
       return <pubtitle> data($t)
              </pubtitle>
let $p_list :=
       for $b in input()/book,
           $p in $b/publisher
       where some $ba in $b/author
             satistfies ($ba is $a)
       return <pubpublisher> data($p)
              </pubpublisher>
return <ref> { $a, $title_list, $p_list}
       </ref>
```

As the system parses the query, it assigns ID qid1 to the first query, and then assigns IDs qid2 and qid3 to the inner queries. During query execution, the system generates invocation IDs for the set of queries; as a result, assuming 10 execution for query block qid2 and 15 executions for query block qid3, the system generates the following list of QueryIDs:

$$\begin{array}{lll} qid1.1 & qid2.1 & qid3.1 \\ & qid2.2 & qid3.2 \\ & \ldots & \ldots \\ & qid2.10 & qid3.10 \\ & & \ldots \\ & & qid3.15 \end{array}$$

■

QueryIDs are used for extending EIDs. An extended EID (ExtEID in the following) is a triple $< QueryID, fileID, Rid >$, where $QueryID$ is the ID of the originating query (0 for persistent elements), $fileID$ is the ID of the file containing the element representation, and Rid is the record id of the element representation. Extended attribute IDs can be defined in the same way.

Beyond uniquely identifying synthetic nodes and distinguishing them from natural nodes, it is necessary to store synthetic nodes in a way allowing the operators of the outer query plan to access and manipulate them. The simplest way to do so is to build for synthetic nodes exactly the same tabular representation as for natural nodes. Synthetic nodes, hence, are represented as records grouped by tag or attribute name, the hierarchical structure being captured by a new Structural Index fragment. Both the node records and the Structural Index fragment are stored in a reserved storage area, distinct from that of the natural nodes or other query nodes (they do not survive the execution of the outer query, so dirtying the representation of persistent nodes has no justification). As a further distinctive feature, synthetic node records are assigned negative fileIDs, obtained by complementing the fileID assigned to their natural counterparts: if book elements are stored in file $fileID1$, then synthetic book elements are stored in file $-fileID1$. For newly created tags or attribute names, new (negative) fileIDs are generated.

For improving the performance of the system during nested query execution, synthetic nodes (as well as their Structural Index fragment) are stored into *virtual* files: these files are allocated in main memory, and flushed to secondary storage only when their size exceeds the capacity of the allocated memory: this policy, similar to the storage policy of hybrid hash join, allows the system to reduce as much as possible the use of secondary storage during query execution.

4.2. Artificial nodes

While synthetic nodes are explicitly stored by the system, artificial nodes (i.e., persistent nodes copied in nested query results) are *shared*, i.e., no new records are created for them, and their EIDs are refreshed by using a special-purpose hash table; this hash table contains one entry for each artificial subtree root, and maps the original EID to the new full QueryID.

Artificial nodes are shared primarily for performance and scalability reasons; without node sharing, artificial nodes should be "duplicated", hence leading to potential unpleasant consequences (e.g., copying the whole database).

Example 4.4 Consider the following simple query:

```
for $b in input()/book
let $ops := for $c in input()/book
            where $b isnot $c
            return $c
return <oddPair> {$b,$ops} </oddPair>
```

This query returns each book in the database, as well as the list of all non-matching book. Without node sharing, executing the inner query would bring to the materialization of nearly the whole database. ■

4.3. Updating positional information

Both synthetic and artificial nodes have their own positions in the nested query result document order. Synthetic node positions are computed in a way close to the position assignment during database creation and indexing, while artificial nodes require a slightly more complex procedure. Since artificial nodes are kept un-materialized, their new positions are obtained by using an offset as shown in Figure 1.

Since each artificial tree has its own offset, the correspondence between tree root EIDs and offsets is kept in a hash table stored in secondary storage (in most case it will be used only during the MergeSort phase, so there is no need to waste precious main memory space). For queries executed under the `UNORDERED` keyword, artificial and synthetic node positions are ignored and not computed.

Since in both cases accessing the positions of artificial nodes is not so cheap (or possible), nested query results can be accessed by using the XMLForwardSeqPath operator only (it uses the Structural Index only); this choice affects only the performance of query execution when inner results are too large to be kept in main memory, so the *trade off* is fully justified.

5. Related works

Most of the work on nested queries is devoted to their logical treatment and manipulation. In [10] author introduced the first algorithm for nested query rewriting in relational databases, along with a classification of such queries. This algorithm was then found incorrect, and alternative algorithms were proposed in [8] and many others.

In [5], authors study the problem of nested query rewriting in object databases. First, they extend Kim's taxonomy of relational nested queries by defining three classification criteria: the *kind of nesting*, i.e., queries of type A, N, J, JA; the *nesting location*, i.e., the presence of nested queries into the $select$, $from$, or $where$ clause of OQL queries; the *kind of dependency*, i.e., the location of references to external variables. Relying on this taxonomy, authors then describe algebraic rewriting rules for most query types.

In [15] author further extends Cluet's approach for supporting a wider range of nested query types.

Rewriting rules for nested queries on XML data, derived from object database rules, are described in [14].

6. Conclusions

This paper describes techniques for efficiently processing and evaluating nested queries on XML data. These techniques, implemented in the Xtasy system, are based on the massive use of node sharing, with the purpose of minimizing the need for intermediate result materialization.

Much work has still to be done: in particular, techniques for correlating results produced by different invocation of the same nested query must be found, in order to reduce evaluation time for queries working on significants fragment of the database.

References

[1] `http://www.di.unipi.it/sartiani/projects/xtasy`.

[2] C. K. Baru, V. Chu, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. Xml-based information mediation for digital libraries. In *Proceedings of the Fourth ACM conference on Digital Libraries, August 11-14, 1999, Berkeley, CA, USA*, pages 214–215. ACM, 1999.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query

Input: a XML tree $\mathcal{T}$, a node creation operation label[$v], an outer node n
Output: a hash table Position Offsets

begin
  if $\$v = t$ then $offset(t) = startpos(n) + 1 - startpos(t)$ fi
  if $\$v = t_1, \ldots, t_n$ then
    begin
      $offset(t_1) = startpos(n) + 1 - startpos(t)$
      $offset(t_2) = endpos(t_1) + 1 - startpos(t_2)$
      ...
      $offset(t_n) = endpos(t_{n-1}) + 1 - startpos(t_n)$
    end
  fi
end

Figure 1. Position update algorithm

Language. Technical report, World Wide Web Consortium, November 2002. W3C Working Draft.

[4] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[5] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical report, University of Karlsruhe, 1994.

[6] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, November 2002. W3C Working Draft.

[7] P. Ferragina, N. Koudasa, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. In *Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.

[8] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 23–33. ACM Press, 1987.

[9] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[10] W. Kim. On optimizing an sql-like nested query. *TODS*, 7(3):443–469, 1982.

[11] A. Malhotra, J. Melton, J. Robie, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Technical report, World Wide Web Consortium, Nov. 2002. W3C Working Draft.

[12] J. M. Martinez. Overview of the mpeg-7 standard (version 5.0). Technical report, International Organisation for Standardisation, 2001.

[13] G. Moerkotte. Incorporating XSL Processing into Database Engines. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002.

[14] C. Sartiani and A. Albano. Yet Another Query Algebra For XML Data. In M. A. Nascimento, M. T. Özsu, and O. Zaïane, editors, *Proceedings of the 6th International Database Engineering and Applications Symposium (IDEAS 2002), Edmonton, Canada, July 17-19, 2002*, 2002.

[15] H. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, 1995.

[16] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating xml. In *SIGMOD 2001*, 2001.

[17] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative xml storage strategies. *SIGMOD Record*, 31(1):5–10, May/June 2002.