XQueC: Pushing Queries to Compressed XML Data

Andrei Arion, INRIA
Angela Bonifati, Icar CNR
Gianni Costa, Icar CNR
Sandra D'Aguanno, INRIA
Ioana Manolescu, INRIA
Andrea Pugliese, Deis UNICAL

1 Introduction

Initially proposed as a data interchange format, XML aims also at becoming a format for data storage and management. However, XML documents in their textual form are rather verbose and tend to predate disk space, due to the textual and repetitive nature of the XML tags and of several XML types.

One solution to this space occupancy problem consists of compressing XML. The XMill project [7] proposed an XML-specific compression method: it compresses the structure (XML tags) separately from the content (data nodes, leaves of the XML tree), which is squeezed into a set of semantically uniform *containers*: for example, one container stores the text values of all <URL> elements in the document, another container stores all <phoneNo> etc. Each container is again separately compressed, by using the best suited compression algorithm; thus, XMill makes maximal use of inherent structure commonalities among semantically similar items. However, an XMill-compressed document is opaque to a query processor: thus, one must fully decompress a full chunk of data before being able to query it. The XGrind project [9] pioneered the field of query processing on compressed XML documents. XGrind does not separate data from structure: an XGrind-compressed XML document is still an XML document, whose tags have been dictionary-encoded, and whose data nodes have been compressed using the Huffmann algorithm [6] and left at their place in the document. XGrind's query processor can be considered an extended SAX parser, which can handle *exact-match and prefix-match queries* on compressed values and *partial-match and range queries* on decompressed values. However, several operations are not supported by XGrind, for example, non-equality selections in the compressed domain. Also, XGrind cannot perform any join, aggregation, nested queries or (construct) operations. Such operations occur in many XML query scenarios, as illustrated by XML benchmarks (e.g., all but the first two of the 20 queries in XMark [8]).

Also, XGrind uses a fixed *root-to-leaf* navigation strategy, which is clearly insufficient to provide for interesting alternative evaluation strategies, as it was done in existing works on querying compressed relational data (e.g., [4], [11]). These works considered evaluating arbitrary SQL queries on compressed data, by comparing (in the traditional framework of cost-based optimization) many query evaluation alternatives, including compression / decompression at several possible points.

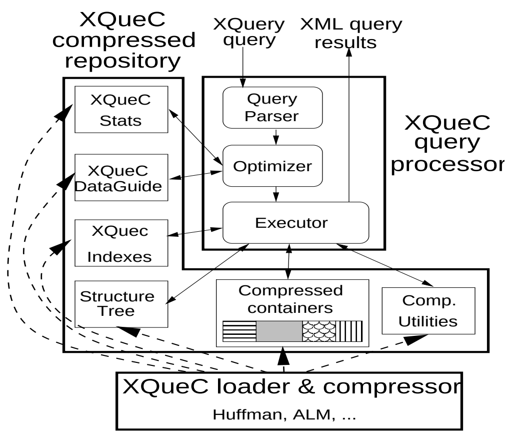

Figure 1: Architecture of the XQueC prototype.

The XQueC system solves all the above-mentioned problems. Our system compresses XML data and queries it as much as possible under its compressed form, covering all real-life, complex classes of queries. The XQueC system adheres to the following approach:

(I) XQueC takes advantage of the XMill principle of compressing separately data and structure for efficiently querying compressed data.

(II) It adopts a simple storage model suitable for compressed XML, and a set of access support structures, allowing for many evaluation alternatives for complex XQuery query. Several storage methods are possible; we view ours as a simple choice for making a proof of concept.

(III) XQueC seamlessly extends a simple algebra for evaluating XML queries to include compression and decompression. This algebra is exploited by a full-fledged cost-based optimizer, choosing a query evaluation method by mixing freely regular operator and compression-relevant ones.

The purpose of the demo is to demonstrate the above features on several XML data sets (see Section 4), among which, the XMark documents [8]. In the following sections, we will use these documents for describing XQueC. A simplified structural outline of these documents is depicted in Figure 2. Each document describes an auction site, with people and open auctions (dashed lines represent IDREFs pointing to IDs and plain lines connect the other XML items).

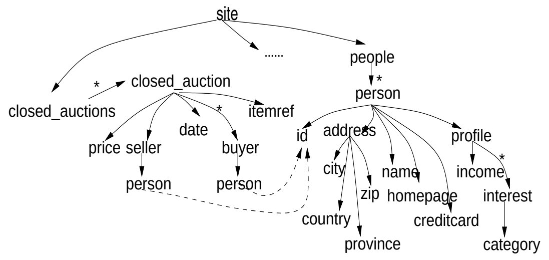

Figure 2: Simplified summary of the XMark XML documents.

Our description of XQueC follows its architecture, depicted in Figure 1. It contains the following modules:
(1) The *loader and compressor* converts XML documents in a compressed, yet queryable format.
(2) The *compressed repository* stores the compressed documents and provides: (i) access methods to this compressed data, and (ii) a set of compression-specific utilities that enable, e.g., the comparison of two compressed values.
(3) The *query processor* optimizes and evaluates XQuery queries over the compressed documents. Its complete set of physical operators allows for efficient evaluation over the compressed repository.

The rest of this paper is organized as follows. In section 2, we motivate the choice of our storage structures for compressed XML, and of the compression algorithms employed. Section 3 describes the XQueC query processor, its set of physical operators, and outline its optimization algorithm. Section 4 briefly presents the demonstration scenario we intend to show.

2 Compressing XML documents in a queryable format

In this section, we first present the principles behind our approach for storing compressed XML documents, and the resulting storage model. Then, we discuss our choice of compression algorithms.

2.1 Compression principles

Our approach for compressing XML was guided by the following principles:

> *Enable a full-fledged algebraic exploration of alternative query evaluation plans (QEPs), including top-down, bottom-up, and direct (index-based) access to a given node in the document.*

To that purpose, we follow the XMill principle of *compressing content separately from structure*. As we will see, this separation provides the basis for alternative QEPs. In particular, value containers fullfill a double role: compressed storage, and access support structure (index).

> *Allow, among other evaluation strategies, processing full XQuery on the compressed documents (also termed* lazy decompression *in [4]).*

This has two consequences. First, w.r.t. XMill, we use a *fine-grained compression* within the container, i.e. compress each leaf data node individually; to that purpose, the *document storage* has to be organized at a fine granularity, as opposed to storing full containers as “blobs”, as done in XMill. While applying fine-grained compression, data items of the same type are compressed using the same algorithm. Thus, we can benefit from the data commonalities and provide the access to each compressed data item at the same time. Fine-grained compression has been proved beneficial also in the context of compressed relational databases [11]. Second, in order to perform comparisons of the form $x_1 < x_2$ without having to decompress x_1 and x_2, we choose to support *order-preserving compression algorithms*, as well as order-agnostic ones (unlike XGrind and XMill). A compression algorithm $comp$ preserves order if for any x_1, x_2, $comp(x_1) < comp(x_2)$ iff $x_1 < x_2$. In the sequel, we will adopt the following short notation ${x_1}^c$ for a compressed value $comp(x_1)$.

2.2 Compressed storage structures

The XQuec loader and compressor parses an XML document and splits it into several data structures, summarized in Figure 1.

Node name dictionary: We use a dictionary to encode the element and attribute names present in an XML document. Thus, if there are N_t distinct names, we assign to each of them a bit string of length $log_2(N_t)$. For example, the XMark documents use 92 distinct names, which we encode on 7 bits, for example:
site: 0000000 | regions: 0000001 | categories: 0000010
Structure tree: We assign to each non-value XML node (element or attribute) an unique integer ID, reflecting the order of nodes in the document. The structure tree is stored as a sequence of *node records*, where each record contains: its own ID, the corresponding tag code; the IDs of its children; and (redundantly) the ID of its parent. For better query performance, as an access support structure, we construct and store a B+ search tree on top of the sequence of node records.
Value containers: All data values found under the same root-to-leaf path expression in the document, like site/open_auctions/ open_auction/interval/start, site/people/person/homepage etc., are stored together into homogeneous containers. In general, we may store in a container values found under several distinct paths, e.g., auction start and end dates. A container is a sequence of *container records*, each one consisting of: a compressed value, and a pointer to parent of this value in the structure tree. Records are placed in the order dictated by the original data values, to enable fast binary search. A sample container instance is:
site/open_auctions/open_auction/interval/start:
(“Jan 01, 2002”)^c | 90 || (“Mar 10, 2002”)^c | 7 ||
(“Aug 05, 2002”)^c | 12 || (“Oct 12, 2002”)^c | 9 ||
We also extend the *node record* of each container entry’s parents with a pointer to the entry, as shown in Figure 3.

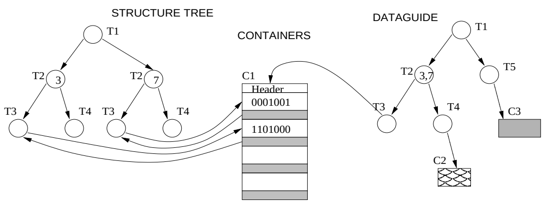

Figure 3: Storage structures in the XQueC repository.

Dataguide: The loader also constructs, as a redundant access support structure, a *strong* dataguide of the XML document. A dataguide [5] is a structural summary representing all possible paths in the document; for tree-structured XML documents, it will always have less nodes than the document (typically by orders of magnitude). A dataguide of the auction documents can be derived from Figure 2, by (i) omitting the dashed edges, which brings it to a tree form, and (ii) storing in each non-leaf node in Figure 3, accessible in this tree by a path p, for example site/people/person, the list of nodes accessible in the XML document by the path p. Finally, the leaf nodes of our dataguide point to the corresponding value containers. Note that while Figure 3 depicts a complete dataguide, in the presence of query workload information, we may prune the parts that are never accessed by queries.

Other indexes and statistics: When loading a document, other indexes and/or statistics can be created, either on the value containers, or on the structure tree. Our loader prototype currently gathers simple fan-out and cardinality statistics (e.g. number of person elements).

Storage alternatives: There are many ways to store XML in general [1]. Any storage mechanism for XML can be seamlessly adopted in XQueC, as long as it allows the presence of containers and the facilities to access container items. Our proposed storage structure is the *simplest and most compact* one that fullfils the principles listed at the beginning of section 2. To measure the occupancy of our structures, we have used a set of documents produced by means of the *xmlgen* generator of the XMark project and ranged from 115KB to 46MB. They have been reduced by a factor ranging from 38% to 45% after compression (these figures include all the above access support structures).

2.3 Container compression algorithms

If we want to enable comparison on compressed values directly, the same compressed algorithm should be used for both items to be compared; therefore, we make the choice of compression algorithms once per data type (assuming only values of similar types are compared; otherwise, decompression and type cast are required).

Compression of numerical attributes is not crucial for XML; in XQueC, we have chosen to encode numbers by means of a simple bit-encoding scheme. String compression instead can strongly impact performances already in the context of database compression and increasingly in the XML context. We had initially three choices for encoding strings in an order-preserving manner: the Arithmetic, Hu-Tucker, and ALM algorithms [2]. Dictionary-based encoding has demonstrated its effectiveness w.r.t. other non-dictionary approaches while ALM has outperformed Hu-Tucker. The former being both dictionary-based and efficient, is a good choice for our system. For order-agnostic compression, we have chosen the non-adaptive version of the Huffmann algorithm [6].

3 Evaluating XML queries over compressed data

The XQueC Query Processor consists of a *query parser*, which is standard, an *optimizer* and *evaluator*. The optimizer uses a regular set of logical operators, and the physical operators which we describe next. Also, the optimizer translates XPath expressions using the $//$ axis into parent-child ones, using our structure dataguide.

3.1 XQueC physical operators

These operators can be divided in three classes: compression and decompression, data access operators and regular operators like $\bowtie$ (join) or σ (selection), which have been programmed to uniformly operate on compressed and on uncompressed data.

Compression / decompression operators: To account for the compression applied on attributes in an XQueC result set, we enhance the column metadata of the result set (in JDBC style) with its *compression status*: a token indicating its compression algorithm (if any), $\emptyset$ otherwise. To modify as desired the compression status of a set of tuples, we use two generic operators: $comp(attrs,\ algo)$ and $decomp(attrs, algo)$, where $attrs$ is a list of attributes present in the operator input, and $algo$ is the compression algorithm to be used. $comp$ performs the compression of attributes of the input tuples, and $decomp$ the opposite.

Data access operators: This library includes the following access operators for the *structure tree*: $root_{acc}(d)$ operator, which, given a document name d, returns the node record of d's root; the $par_{acc}(ListID)$ and $child_{acc}(ListID)$ operators take as input a list of node IDs and, using the structure tree and the B+ tree index on them, return the node records of the children, resp. parents of the input nodes; the $dg_{acc}(d,p)$ operator takes as parameters the name of a document d and a path p and returns the list of pointers to the node records for the elements/attributes found in d under path p (indexed access). Then, there are two specialized operators for the *containers:* an operator that makes a sequential scan of the container , $cont_{scan}(p)$, which accesses the container records identified by the root-to-leaf path expression p; the $cont_{acc}(p,\ \theta,\ x^c)$ operator that takes as parameters a root-to-leaf parent-child path expression p, e.g., auction/people/person/name, a comparison operator θ like $=$, $\neq$, $<$, $\leq$, and a compressed value x^c. The result of $cont_{acc}(p,\ \theta,\ x^c)$ is the set of container records associated to p, whose encapsulated compressed values val^c satisfy $val^c\ \theta\ x^c$. If the compression on the container for p preserves order, then x^c can be directly used as a search key (when θ is $=, <, \leq$) to find the qualifying records. Thus, with order-preserving compression, the containers act like indexes, allowing for efficient search.

Finally, the $val_{acc}(cr)$ operator takes as input a set of container records cr, and (following their included parent pointers) returns the node records whose values are in cr.

Comparison operators on compressed data: this library contains the comparison operators ($\bowtie, \sigma$), which are able to work uniformly on compressed and uncompressed inputs; it is the task of the optimizer to (i) determine which one to use and (ii) make sure that the proper compression / decompression steps have been taken so that the attributes to be compared by $\bowtie$ or σ have the same compression status. We implement a simple, pipelined σ physical operator, and two flavors of join: sort-merge $\bowtie_{sort}$, and hash $\bowtie_{hash}$.

Examples of XMark QEPs As an example, consider query Q_1 from XMark:

```
FOR $b IN document("auction.xml")/site/people/
        person[id="person0"]
RETURN $b/name/text()
```

For the sake of clarity, when illustrating query plans we omitted the class acc of the operator. Figure 4 shows three QEPs for Q_1, applying lazy decompression; to do more work on uncompressed data, one only needs to push down some $decomp$ operators. The first one starts from id elements and then navigates up and down to person and name elements. We compress the tags appearing in Q_1 to their dictionary encoding, and we give them as parameters to σ_{tag}'s whenever we need to test an element's tag. The plan in Figure 4(b) starts from the name container. Finally, the third one starts from both containers and performs a join. Since id is an unique key for persons, the plan in Figure 4(a) is likely to be the best one, and will be identified as such by the optimizer.

4 Demonstration Highlights

Our demonstration addresses three main issues, regarding the compressor and the query processor. **Compressor:** we will first show the compression ratio and compression time of different kinds of data, from the regular schema-driven ones to the irregular ones, with heavy textual content. We have implemented ALM and Huffman [6] and we will show figures for both algorithms, revealing that they are as good as those for unqueryable compressor (i.e. XMill). **Query Processor (1):** we will pinpoint the querying time for the queries of the XML benchmark, and compare them with the querying time of the same query while executed in a compression-unaware query processor and in XGrind (only for the queries of the benchmark supported by the latter). **Query Processor (2):** we will draw the querying time for several XQuery Use Cases [10] and for some of the novel XPath and XQuery Full-Text Use Cases [10]. In particular, for the latter we realized that by using ALM, we can nearly address all the proximity, wildcards and fuzzy-matching queries.

In any of these cases, the search for a word within a statement is done in the compressed domain and needs less time than in the uncompressed one. As a future development, we plan to integrate in our framework more sophisticated textual indexes, which should further improve performances.

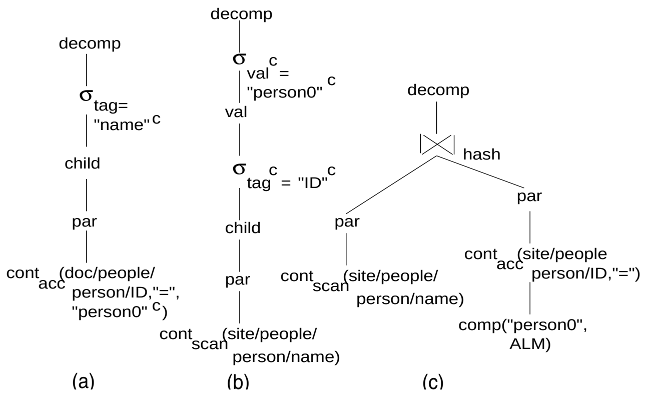
decomp
σ tag="name"c
child
par
cont acc(doc/people/person/ID,"=","person0"c)
(a)
decomp
σ val="person0"c
val
σ tag="ID"c
child
par
cont scan(site/people/person/name)
(b)
decomp
hash
par
cont scan(site/people/person/name)
par
cont acc(site/people person/ID,"=")
comp("person0", ALM)
(c)

Figure 4: Sample QEPs for XMark's Q_1.

All the implementation (compressor and query processor) of XQueC is done in Java. We use Berkeley DB [3] to implement our in-house storage system. Berkeley DB Data Store provides a library of elementary database structure (hash tables, B+ trees etc.) on top of which any desired storage can be configured. For our specific storage model, Berkeley DB seemed to be the most performant, reliable, and flexible choice for compressed data storage.

The demonstration will be shown on a PC under Linux. We will use different kinds of synthetic and real XML data. Besides XMark, we plan to demonstrate it on the DBLP data and (regular or schema-driven data) Health Level 7 dataset and (irregular data) Library of Congress dataset.

References

[1] S. Amer-Yahia and M. Fernandez. Overview of existing XML storage techniques. submitted for publication, 2002.

[2] Gennady Antoshenkov. Dictionary-based order-preserving string compression. *VLDB Journal*, 6(1):26–39, 1997.

[3] Berkeley DB Data Store. `http://www.sleepycat.com/products/data.shtml/`.

[4] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *Proc. of ACM SIGMOD Conf.*, 2000.

[5] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Int'l VLDB Conf.*, pages 436–445, 1997.

[6] D. A. Huffman. A Method for Construction of Minimum-Redundancy Codes. In *Proc. of the IRE*, 1952.

[7] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *Proc. of ACM SIGMOD Conf.*, 2000.

[8] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *Proc. of the Int'l VLDB Conf.*, 2002.

[9] P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. of the ICDE Conf.*, 2002.

[10] XQuery (and XPath Full-text) Use Cases. `http://www.w3.org/TR/xmlquery(-full-text)-use-cases/`.

[11] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.