Efficient query evaluation over compressed XML data

Andrei Arion[1], Angela Bonifati[2], Gianni Costa[2], Sandra D'Aguanno[1], Ioana Manolescu[1], and Andrea Pugliese[3]

[1] INRIA Futurs - LRI,
4 rue Jean Monod, 91893 Orsay Cedex, France
{firstname.lastname}@inria.fr
[2] Icar-CNR, Via P. Bucci 41/C,
87036 Rende (CS), Italy
{bonifati, costa}@icar.cnr.it
[3] DEIS, University of Calabria, Via P. Bucci 41/C,
87036 Rende(CS), Italy
apugliese@si.deis.unical.it

Abstract. XML suffers from the major limitation of high redundancy. Even if compression can be beneficial for XML data, however, once compressed, the data can be seldom browsed and queried in an efficient way. To address this problem, we propose *XQueC*, an [*XQue*]ry processor and [*C*]ompressor, which covers a large set of XQuery queries in the compressed domain. We shred compressed XML into suitable data structures, aiming at both reducing memory usage at query time and querying data while compressed. XQueC is the first system to take advantage of a query workload to choose the compression algorithms, and to group the compressed data granules according to their common properties. By means of experiments, we show that good trade-offs between compression ratio and query capability can be achieved in several real cases, as those covered by an XML benchmark. On average, XQueC improves over previous XML query-aware compression systems, still being reasonably closer to general-purpose query-unaware XML compressors. Finally, QETs for a wide variety of queries show that XQueC can reach speed comparable to XQuery engines on uncompressed data.

1 Introduction

XML documents have an inherent textual nature due to repeated tags and to PCDATA content. Therefore, they lend themselves naturally to compression. Once the compressed documents are produced, however, one would like to still query them under a compressed form as much as possible (reminiscent of "lazy decompression" in relational databases [1], [2]). The advantages of processing queries in the compressed domain are several: first, in a traditional query setting, access to small chunks of data may lead to less disk I/Os and reduce the query processing time; second, the memory and computation efforts in processing compressed data can be dramatically lower than those for uncompressed ones, thus even low-battery mobile devices can afford them; third, the possibility of obtaining compressed query results allows to spare network bandwidth when sending these results to a remote location, in the spirit of [3].

Previous systems have been proposed recently, i.e. XGrind [4] and XPRESS [5], allowing the evaluation of simple path expressions in the compressed domain. However, these systems are based on a naive top-down query evaluation mechanism, which is not enough to execute queries efficiently. Most of all, they are not able to execute a large set of common XML queries (such as joins, inequality predicates, aggregates, nested queries etc.), without spending prohibitive times in decompressing intermediate results.

In this paper, we address the problem of compressing XML data in such a way as to allow efficient XQuery evaluation in the compressed domain. We can assert that our system, XQueC, is the first XQuery processor on compressed data. It is the first system to achieve a good trade-off among data compression factors, queryability and XQuery expressibility. To that purpose, we have carefully chosen a fragmentation and storage model for the compressed XML documents, providing selective access paths to the XML data, and thus further reducing the memory needed in order to process a query. The XQueC system has been demonstrated at VLDB 2003 [6].

The basis of our fragmentation strategy is borrowed from the XMill [7] project. XMill is a very efficient compressor for XML data, however, it was not designed to allow querying the documents under their compressed form. XMill made the important observation that data nodes (leaves of the XML tree) found on the same path in an XML document (e.g. /site/people/person/address/city in the XMark [8] documents) often exhibit similar content. Therefore, it makes sense to group all such values into a single *container* and choose the compression strategy *once per container*. Subsequently, XMill treated a container like a single "chunk of data" and compressed it as such, which disables access to any individual data node, unless the whole container is decompressed. Separately, XMill compressed and stored the structure tree of the XML document.

While in XMill a container may contain leaf nodes found under several paths, leaving to the user or the application the task of defining these containers, in XQueC the fragmentation is always dictated by the paths, i.e., we use one container per root-to-leaf path expression. When compressing the values in the container, like XMill, we take advantage of the commonalities between all container values. But most importantly, unlike XMill, *each container value is individually compressed and individually accessible*, enabling an effective query processing.

We base our work on the principle that XML compression (for saving disk space) and sophisticated query processing techniques (like complex physical operators, indexes, query optimization etc.) can be used together when properly combined. This principle has been stated and forcefully validated in the domain of relational query processing [1], [3]. Thus, it is not less important in the realm of XML.

In our work, we focus on the right compression of the *values* found in an XML document, coupled with a compact storage model for all parts of the document. Compressing the *structure* of an XML document has two facets. First, XML tags and attribute names are extremely repetitive, and practically all systems (indeed, even those not claiming to do "compression") encode such tags by means of much more compact tag numbers. Second, an existing work [9] has addressed the summarization of the tree structure itself, connecting among them parent and child nodes. While structure compression is interesting, its advantages are not very visible when considering the XML document as a whole. Indeed, for a rich corpus of XML datasets, both real and synthetic, our measures have shown that values make up 70% to 80% of the document structure. Projects like XGrind [4] and XPRESS [5] have already proposed schemes for value compression that would enable querying, but they suffer from limited query evaluation techniques (see also Section 1.2). These systems apply a fixed compression strategy regardless of the data and query set. In contrast, our system increases the compression benefits by adapting its compression strategy to the data and query workload, based on a suitable cost model.

By doing data fragmentation and compression, XQueC indirectly targets the problem of main-memory XQuery evaluation, which has recently attracted the attention of the community [9], [10]. In [10], the authors show that some current XQuery prototypes are in practice limited by their large memory consumption; due to its small footprint, XQueC scales better (see Section 5). Furthermore, some such in-memory prototypes exhibit prohibitive query execution times even for simple lookup queries. [9] focuses on the problem of fitting into memory a narrowed version of the tree of tags, which is however a small percentage of the overall document, as explained above.

XQueC addresses this problem in a two-fold way. First, in order to diminish its footprint, it applies powerful compression to the XML documents. The compression algorithms that we use allow to evaluate most predicates directly on the compressed values. Thus, decompression is often necessary only at the end of the query evaluation (see Section 4). Second, the XQueC storage model includes lightweight access support structures for the data itself, providing thus efficient primitives for query evaluation.

1.1 The XQueC system

The system we propose compresses XML data and queries them as much as possible under its compressed form, covering all real-life, complex classes of queries.

The XQueC system adheres to the following principles:

1. As in XMill, data is collected into containers, and the document structure stored separately. In XQueC, there is a container for each different $< type, pe >$, where pe is a distinguished root-to-leaf path expression and type is a distinguished elementary type. The set of containers is then partitioned again to allow for better sharing of compression structures, as explained in Section 2.2.
2. In contrast with previous compression-aware XML querying systems, whose storage was plainly based on files, XQueC is the first to use a complete and robust storage model for compressed XML data, including a set of access support structures. Such storage is fundamental to guarantee a fast query evaluation mechanism.
3. XQueC seamlessly extends a simple algebra for evaluating XML queries to include compression and decompression. This algebra is exploited by a cost-based optimizer, which may choose query evaluation strategies, that freely mix regular operator and compression-aware ones.
4. XQueC is the first system to exploit the *query workload* to *(i)* partition the containers into sets according to the source model[4] and to *(ii)* properly assign the most suitable compression algorithm to each set. We have devised an appropriate cost model, which helps making the right choices.

[4] The source model is the model used for the encoding, for instance the Huffman encoding tree for Huffman compression [11] and the dictionary for ALM compression [12], outlined later.

5. XQueC is the first compressed XML querying system to use the order-preserving[5] textual compression. Among several alternatives, we have chosen to use the ALM [12] compression algorithm, which provides good compression ratios and still allows fast decompression, which is crucial for an algorithm to be used in a database setting [13]. This feature enables XQueC to evaluate, in the compressed domain, the class of queries involving inequality comparisons, which are not featured by the other compression-aware systems.

In the following sections, we will use XMark [8] documents for describing XQueC. A simplified structural outline of these documents is depicted in Figure 1 (at right). 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). We describe XQueC following its architecture, depicted in Figure 1 (at left). It contains the following modules:

1. The *loader and compressor* converts XML documents in a compressed, yet queryable format. A cost analysis leverages the variety of compression algorithms and the query workload predicates to decide the partition of the containers.
2. The *compressed repository* stores the compressed documents and provides: *(i)* compressed data access methods, and *(ii)* a set of compression-specific utilities that enable, e.g., the comparison of two compressed values.
3. The *query processor* evaluates XQuery queries over compressed documents. Its complete set of physical operators (regular ones and compression-aware ones) allows for efficient evaluation over the compressed repository.

1.2 Related Work

XML data compression was first addressed by XMill [7], following the principles outlined in the previous section. After coalescing all values of a given container into a single data chunk, XMill compresses separately each container with its most suited algorithm, and then again with `gzip` to shrink it as much as possible. However, an XMill-compressed document is opaque to a query processor: thus, one must fully decompress a whole chunk of data before being able to query it.

The XGrind system [4] aims at query-enabling XML compression. 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 Huffman [11] algorithm 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. Therefore, 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 naive top-down navigation strategy, which is clearly insufficient to provide for interesting alternative evaluation strategies, as it was done

[5] Note that a compression algorithm $comp$ preserves order if for any x_1, x_2, $comp(x_1) < comp(x_2)$ iff $x_1 < x_2$.

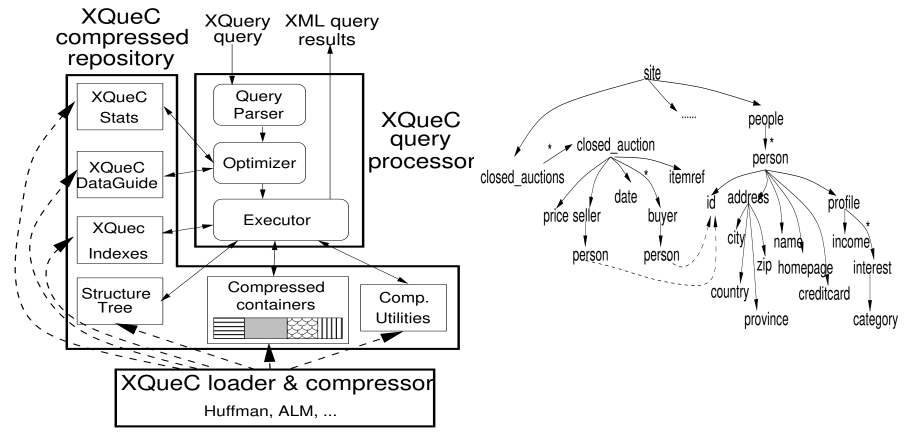


Fig. 1. Architecture of the XQueC prototype (left); simplified summary of the XMark XML documents (right).

in existing works on querying compressed relational data (e.g., [1], [2]). 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.

A third recent work, XPRESS [5] uses a novel *reverse arithmetic encoding* method, mapping entire path expressions to intervals. Also, XPRESS uses a simple mechanism to infer the type (and therefore the compression method suited) of each elementary data item. XPRESS's compression method, like XGrind's, is homomorphic, i.e. it preserves the document structure.

To summarize, while XML compression has received significant attention [4], [5], [7], querying compressed XML is still in its infancy [4], [5]. Current XML compression and querying systems do not come anywhere near to efficiently executing complex XQuery queries. Indeed, even the evaluation of XPath queries is slowed down by the use of the fixed top-down query evaluation strategy.

Moreover, the interest towards compression even in a traditional data warehouse setting is constantly increasing in commercial systems, such as Oracle [14]. In [14], it is shown that the occupancy of raw data can be reduced while not impacting query performance. In principle, we expect that in the future a big share of this data will be expressed in XML, thus making the problem of compression very appealing.

Finally, for what concerns information retrieval systems, [15] exploits a variant of Huffman (extended to "bytes" instead of bits) in order to execute phrase matching entirely in the compressed domain. However, querying the text is obviously only a subset of the XQuery features. In particular, theta-joins are not feasible with the above variant of Huffman, whereas they can be executed by means of order-aware ALM.

1.3 Organization

The paper is organized as follows. In Section 2, we motivate the choice of our storage structures for compressed XML, and present ALM [12] and other compression algorithms, which we use for compressing the containers. Section 3 outlines the cost model used for partitioning the containers into sets, and for identifying the right compression to be applied to the values in each container set. Section 4 describes the XQueC query processor, its set of physical operators, and outlines its optimization algorithm. Section 5 shows the performance measures of our system on several data sets and XQuery queries.

2 Compressing XML documents in a queryable format

In this section, we present the principles behind our approach for storing compressed XML documents, and the resulting storage model.

2.1 Compression principles

In general, we make the observation that within XML text, strings represent a large percentage of the document, while numbers are less frequent. Thus, compression of strings, when effective, can truly reduce the occupancy of XML documents. Nevertheless, not all compression algorithms can seamlessly afford string comparisons in the compressed domain. In our system, we include both order-preserving and order-agnostic compression algorithms, and the final choice is entrusted to a suitable cost model.

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

Order-agnostic compression. As an order-agnostic algorithm, we chose classical Huffman [6], which is universally known as a simple algorithm which achieves the best possible redundancy among the resulting codes. The process of encoding and decoding is also faster than universal compression techniques. Finally, it has a set of fixed codewords, thus strings compressed with Huffman can be compared in the compressed domain within equality predicates. However, inequality predicates need to be decompressed. That is why in XQueC we may exploit order-preserving compression as well as not order-preserving one.

Order-preserving compression. Whereas everybody knows the potentiality of Huffman, the choice of an order-preserving algorithm is not immediate. We had initially three choices for encoding strings in an order-preserving manner: the Arithmetic [16], Hu-Tucker [17] and ALM [12] algorithms. We knew that dictionary-based encoding has demonstrated its effectiveness w.r.t. other non-dictionary approaches [18] while ALM has outperformed Hu-Tucker (as described in [19]). The former being both dictionary-based and efficient, was a good choice in our system. ALM has been used in relational databases for blank-padding (i.e. in Oracle) and for indexes compression. Due to its dictionary-based nature, ALM decompresses faster than Huffman, since it outputs bigger portions of a string at a time, when decompressing. Moreover, ALM seamlessly solved the problem of order-preserving dictionary compression, raised by encodings such as Zilch encoding, string prefix compression and composite key compression by improving each of these. To this purpose, ALM eliminates the prefix property exhibited by those former encodings by allowing in the dictionary more than one symbol for the same prefix.

[6] Here and in the remainder of the paper, by Huffman we shall mean solely the classical Huffman algorithm [11], thus disregarding its variants.

Token	*Code*	*Interval*
.....	....	
the	c	[theaa, therd]
there	d	[there, there]
the	e	[therf, thezz]
ir	b	[ir, ir]
.....	....	
se	v	[se, se]

String	*Code*
their	cb
there	d
these	ev

Fig. 2. An example of encoding in ALM.

We now provide a short overview of how the ALM algorithm works. The fundamental mechanics behind the algorithm tells to consider the original set of source substrings, to split it into disjunct partitioning intervals set and to associate an interval prefix to each partitioning interval. For example, Figure 2 shows the mapping from the original source (made of the strings `there, their, these`) into some partitioning intervals and associated prefixes, which clearly do not scramble the original order among the source strings. We have implemented our own version of the algorithm, and we have obtained encouraging results w.r.t. previous compression-aware XML processors (see Section 5).

Workload-based choices of compression. Among the possible predicates writable in an XQuery query, we distinguish among the inequality, equality and wildcard. The ALM algorithm [12] allows inequality and equality predicates in the compressed domain, but not wildcards, whereas Huffman [11] supports prefix-wildcards and equality but not inequality. Thus, the choice of the algorithm can be aided by a proper query workload, whenever this turns to be available. In case, instead, the workload has not been provided, XQueC uses ALM for strings and decompresses the compared values in case of wildcard operations.

Structures for algebraic evaluation. Containers in XQueC closely resemble B+trees on values. Moreover, a light-weight structure summary allows for accessing the structure tree and the data containers in the query evaluation process. Data fragmentation allows for better exploiting all the possible evaluation plans, i.e. bottom-up, top-down, hybrid or index-based. As shown below, several queries of the XMark benchmark take advantage of the XQueC appropriate structures and of the consequent flexibility in parsing and querying these compressed structures.

2.2 Compressed storage structures

The XQueC loader/compressor parses and splits an XML document into the data structures depicted 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.

Structure tree. We assign to each non-value XML node (element or attribute) an unique integer ID. 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

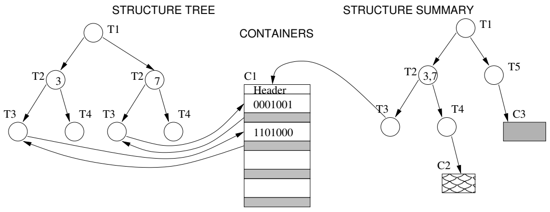


Fig. 3. Storage structures in the XQueC repository

node records. Finally, each node record points to all its attribute and text children in their respective containers.

Value containers. All data values found under the same root-to-leaf path expression in the document are stored together into homogeneous containers. 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 not placed in the document order, but in a lexicographic order, to enable fast binary search. Note that container generation as done in XQueC is reminiscent of vertical partitioning of relational databases [20]. This kind of partitioning guarantees random access to the document content at different points, i.e. the containers access points. This choice provides interesting query evaluation strategies and leads to good query performance (see Section 5). Moreover, containers, even if kept separated, may share the same source model or, they can be compressed with different algorithms if not involved in the same queries. This is decided by a cost analysis which exploits the query workload and the similarities among containers, as described in Section 3.

Structure summary. The loader also constructs, as a redundant access support structure, 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 several orders of magnitude). A structural summary of the auction documents can be derived from Figure 1, 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, the list of nodes reachable in the document instance by the same path. Finally, the leaf nodes of our structure summary point to the corresponding value containers. Note that the structure summary is very small, thus it does not affect the compression rate. Indeed, in our experiments on the corpus of XML documents described in Section 5, the structure summary amounts to about 19% of the original document size.

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).

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 an average factor of 60% after compression (these figures include all the above access structures).

Our proposed storage structure is the simplest and most compact one that fulfills the principles listed at the beginning of Section 2; there are many ways to store XML in general [21]. If we omit our access support structures (backward edges, B+ index,

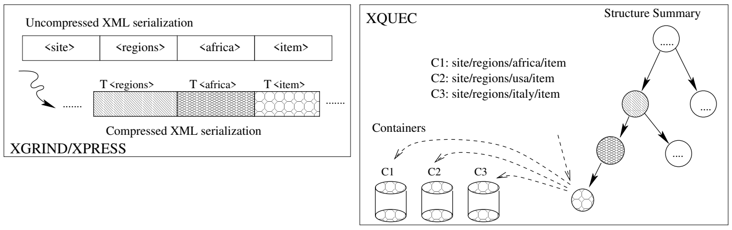


Fig. 4. Accesses to containers in case of XMark's Q14 with *descendant-or-self* axis in XPress/XGrind versus XQueC.

and the structure summary), we shrink the database by a factor of 3 to 4, albeit at the price of deteriorated query performance.

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.

2.3 Memory issues

Data fragmentation in XQueC guarantees a wide variety of query evaluation strategies, and not solely top-down evaluation as in homomorphic compressors [4], [5]. Instead of identifying at compile-time the parts of the documents necessary for query evaluation, as given by an XQuery projection operator [10], in XQueC the path expressions are hard-coded into the containers and projection is already prepared in advance when compressing the document, without any additional effort for the loader. Consider as examples the following query Q14 of XMark:

```
FOR $i IN document("auction.xml")/site//item
WHERE CONTAINS($i/description,"gold")
RETURN $i/name/text()
```

This query would require prohibitive parsing times in XGrind and XPRESS, which basically have to load into main-memory all the document and parse it entirely in order to find the sought items. For this query, as shown in Figure 4, all the XML stream has to be parsed to find the elements `<item>`.

In XQueC, the compressor has already shredded the data and accessibility to these data from the structure summary allows to save the parsing and loading times. Thus, in XQueC the structure summary is parsed (not all the structure tree), then the involved containers are directly accessed (or alternatively their selected single items) and loaded into main-memory. More precisely, as shown in Figure 4, once the structure summary leads to the containers C_1, C_2 and C_3, only these (or part of them) need to be fetched in memory. Finally, note that in Galax, extended with the projection operator [10], the execution times for queries involving the *descendant-or-self* axis (such as XMark Q14) are significantly increased, since additional complex computation is demanded to the loader for those queries.

3 Compression choices

XQueC exploits the query workload to choose the way containers are compressed. As already highlighted, the containers are filled up with textual data, which represents a big share of the whole documents. Thus, achieving a good trade-off between compression ratio and query execution times, must necessarily imply the capability to make a good choice for textual container compression.

First, a container may be compressed with any compression algorithm, but obviously one would like to apply a compression algorithm with nice properties. For instance, the decompression time for a given algorithm strongly influences the times of queries over data compressed with that algorithm. In addition, the compression ratio achieved by a given algorithm on a given container influences the overall compression ratio.

Second, a container can be compressed separately or can share the same source model with other containers. The latter choice would be very convenient whenever for example two containers exhibit data similarities, which will improve their common compression ratio. Moreover, the occupancy of the source model is as relevant in the choice of the algorithm as the occupancy of containers.

To understand the impact of compression choices, consider two binary-encoded containers, ct_1 and ct_2. ct_1 contains only strings composed of letters `a` and `b`, whereas ct_2 contains only strings composed of letters `c` and `d`. Suppose, as one extreme case, that two separate source models are built for the two containers; in such a case, containers are encoded with 1 bit per letter. As the other extreme case, a common source model is used for both containers, thus requiring 2 bits per letter for the encoding, and increasing the containers occupancy. This scenario may get even more complicated when we think of an arbitrary number of encodings assigned to each container. This smallish example already shows that compressing several containers with the same source model leads to losses in the compression ratio.

In the sequel, we show how our system addresses these problems, by proposing a suitable cost model, a greedy algorithm for making the right choice, and some experimental results. The cost model of XQueC is based on the set of non-numerical (textual) containers, the set of available compression algorithms $\mathcal{A}$, and the query workload $\mathcal{W}$. As it is typical of optimization problems, we will characterize the search space, define the cost function, and finally propose a simple search strategy.

3.1 Search space: possible compression configurations

Let $\mathcal{C}$ be the set of containers built from a set of documents $\mathcal{D}$. A *compression configuration* s for $\mathcal{D}$ is denoted by a tuple $<P, alg>$ where P is a partition of $\mathcal{C}$'s elements, and the function $alg : P \rightarrow \mathcal{A}$ associates a compression algorithm with each set p in the partition P. The configuration s dictates thus that all values of the containers in p will be compressed using $alg(p)$, and a single common source model. Moreover, let $\mathcal{P}$ be the set of possible partitions of $\mathcal{C}$. The cardinality of $\mathcal{P}$ is the *Bell number* $B_{|\mathcal{C}|}$, which is exponential with $|\mathcal{C}|$. For each possible partition $P_i \in \mathcal{P}$, there are $|\mathcal{A}|^{|P_i|}$ ways of assigning a compression algorithm to each set in P_i. Therefore, the size of the search space is: $\sum_{i=1}^{B_{|C|}} |\mathcal{A}|^{|P_i|}$, which is exponential in $|\mathcal{A}|$.

3.2 Cost function: appreciating the quality of a compression configuration

Intuitively, the cost function for a configuration s reflects the time needed to apply the necessary data decompressions in order to evaluate the predicates involved in the queries of $\mathcal{W}$. Reasonably, it also accounts for the compression ratios of the employed compression algorithms, and it includes the cost of storing the source model structures. The cost of a configuration s is an integer value computed as a weighted sum of storage and decompression costs.

Characterization of compression algorithms. Each algorithm $a \in \mathcal{A}$ is denoted by a tuple $< d_c, c_s(F), c_a(F),\ eq,\ ineq,\ wild >$. The *decompression cost* d_c is an estimate of the cost of decompressing a container record by using a, the *storage cost* $c_s(F)$ is a function estimating the cost of storing a container record compressed with a, and the *storage cost of the source model structures* $c_a(F)$ is a function estimating the cost of storing the source model structures for a container record. F is a symmetric *similarity matrix* whose generic element $F[i,\ j]$ is a real number ranging between 0 and 1, capturing the normalized similarity between a container ct_i and a container ct_j. F is built on the basis of data statistics, such as the number of overlapping values, the character distribution within the container entries, and possibly other type information, whenever available (e.g. the XSchema types, using results presented in [22]) [7]. Finally, the *algorithmic properties* eq, $ineq$ and $wild$ are boolean values indicating whether the algorithm supports in the compressed domain: (*i*) equality predicates without prefix-matching (eq), (*ii*) inequality predicates without prefix-matching ($ineq$) and (*iii*) equality predicates with prefix-matching ($wild$). For instance, Huffman will have $eq = true$, $ineq = false$ and $wild = true$, while ALM will have $eq = true$, $ineq = true$ and $wild = false$. We denote each parameter of algorithm a with an array notation, e.g., $a[eq]$.

Storage costs. The containers and source model storage costs are simply computed as $\sum_{p\in P}\left(alg(p)[\widehat{c}(F_p)] * \sum_{c\in p}|c|\right)$ where $\widehat{c} = c_s$ for the case of container storage and $\widehat{c} = c_a$ for source model storage[8]. Obviously, c_s and c_a need not to be evaluated on the overall F but solely on F_p, that is the projection of F over the containers of the partition p.

Decompression cost. In order to evaluate the decompression cost associated with a given compression configuration s, we define three square matrices, E, I and D, having size $(|\mathcal{C}| + 1) \times (|\mathcal{C}| + 1)$. These matrices reflect the comparisons (equality, inequality and prefix-matching equality comparisons, respectively) made in $\mathcal{W}$ among container values or between container values and constants. More formally, the generic element $E_{i,j}$, with $i \neq |\mathcal{C}| + 1$ and $j \neq |\mathcal{C}| + 1$, is the number of equality predicates in $\mathcal{W}$ between ct_i and ct_j not involving prefix-matching, whereas with $i = |\mathcal{C}| + 1$ or $j = |\mathcal{C}| + 1$, it is the number of equality predicates in $\mathcal{W}$ between ct_i and a constant (if $j = |\mathcal{C}| + 1$), or between ct_j and a constant (if $i = |\mathcal{C}| + 1$), not involving prefix-matching. Matrices I and D have the same structure but refer to inequality and prefix-matching comparisons, respectively. Obviously, E, I and D are symmetric.

Considering the generic element of the three matrices, say $M[i,\ j]$, the associated decompression cost is obviously zero if ct_i and ct_j share the same source model and the algorithm they are compressed with supports the corresponding predicate in the compressed domain. A decompression cost occurs in three cases: (*i*) ct_i and ct_j are

[7] We do not delve here into the details of F as study of similarity among data is outside the scope of this paper.

[8] We are not considering here the containers that are not involved in any query in W. Those do not incur a cost so they can be disregarded in the cost model.

compressed using different algorithms; (*ii*) ct_i and ct_j are compressed using the same algorithm but different source models; (*iii*) ct_i and ct_j share the same source model but the algorithm does not support the needed comparison (equality in the case of E, inequality for I and prefix-matching for D) in the compressed domain. For instance, for a generic element $I[i, j]$, in the case of $i \neq j$, $i \neq |\mathcal{C}| + 1$ and $j \neq |\mathcal{C}| + 1$, the cost would be:

- zero, if $ct_i \in p$, $ct_j \in p$, $alg(p)[ineq] = true$;
- $|ct_i| * alg(p')[d_c] + |ct_j| * alg(p'')[d_c]$, if $ct_i \in p'$, $ct_j \in p''$, $p' \neq p''$ (cases (*i*) and (*ii*));
- $(|ct_i| + |ct_j|) * alg(p)[d_c]$, if $ct_i \in p$, $ct_j \in p$, $alg(p)[ineq] = false$ (case (*iii*)).

The decompression cost is calculated by summing up the costs associated with each element of the matrices E, I, and D. However, note that (*i*) for the cases of E and D, we consider $alg(p)[eq]$ and $alg(p)[wild]$, respectively, and that (*ii*) the term referring to the cardinality of the containers to be decompressed is adjusted in the cases of self-comparisons (i.e. $i = j$) and comparisons with constants ($i = |\mathcal{C}| + 1$ or $j = |\mathcal{C}| + 1$).

3.3 Devising a suitable search strategy

XQueC currently uses a greedy strategy for moving into the search space. The search starts with an initial configuration $s_0 =< P_0, alg_0 >$, where P_0 is a partition of $\mathcal{C}$ having sets of exactly one container, and alg_0 blindly assigns to each set a generic compression algorithm (e.g. `bzip`) and a separate source model. Next, s_0 is gradually improved by a sequence of *configuration moves*.

Let $Pred$ be the set of value comparison predicates appearing in $\mathcal{W}$. A move from $s_k =< P_k, alg_k >$ to $s_{k+1} =< P_{k+1}, alg_{k+1} >$ is done by first randomly extracting a predicate $pred$ from $Pred$. Let ct_i and ct_j be the containers involved in $pred$ (for instance $pred$ makes an equality comparison, such as $ct_i = ct_j$, or an inequality one, such as $ct_i > ct_j$). Let p' and p'' the sets in P_k to which ct_i and ct_j belong, respectively. If $p' = p''$, we build a new configuration s' where $alg_{k+1}(p')$ is such that the evaluation of $pred$ is enabled on compressed values, and alg_{k+1} has the greatest number of algorithmic properties holding $true$. Then, we evaluate the costs of s_k and s', and let s_{k+1} be the one with the minimum cost. In the case of $p' \neq p''$, we build two new configurations s' and s''. s' is obtained by dropping ct_i and ct_j from p' and p'', respectively, and adding the set $\{ct_i, ct_j\}$ to P_{k+1}. s'' is obtained by replacing p' and p'' with their union. For both s' and s'', alg_{k+1} associates to the new sets in P_{k+1} an algorithm enabling the evaluation of $pred$ in the compressed domain and having the greatest number of algorithmic properties holding $true$. Finally, we evaluate the costs of s_k, s' and s'', and let s_{k+1} be the one with the minimum cost.

Example. To give a flavor of the savings gained with partitioning the set of containers, consider an initial configuration, which has five containers on an XMark document, all of them sized about 6MB, which we initially (naively) choose to compress with ALM only; let us call this configuration NaiveConf. The workload is made of XQuery queries with inequality predicates over the path expressions leading to the above containers. The first three containers are filled with Shakespeare's sentences,

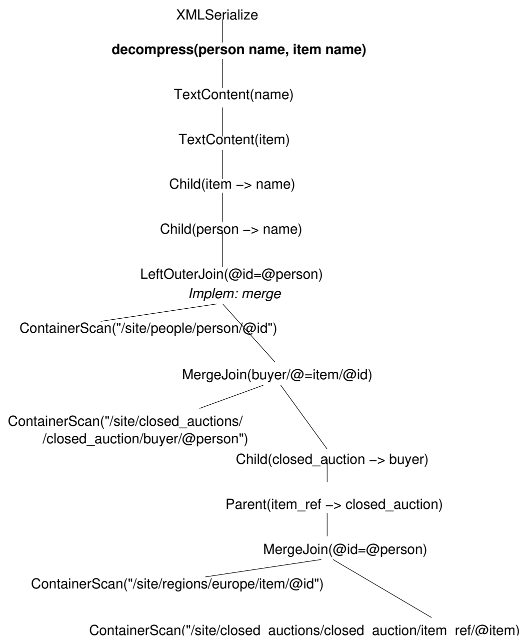


Fig. 5. Query execution plan for XMark's Q9.

the fourth is filled with person names and the fifth with dates. Using the above workload, we obtain the best partitioning, which has three partitions, one with the first three containers and a distinct partition for the fourth and fifth, let us call it *GoodConf*. The compression factor shifts from 56.14% for the *NaiveConf* to 67.14%, 71.75% and 65.15% respectively for the three partitions of *GoodConf*. While in such a case the source models sizes do not vary significantly, the decompression cost in *Good Conf* is clearly advantageous w.r.t. *NaiveConf*, leading to gain 21.42% for shakespearian text, 28.57% for person names and to loose only 6% for dates. □

Note that, for each predicate in $Pred$, the strategy explores a fixed subset of possible configuration moves, so its complexity is linear in $|Pred|$. Of course, due to this partial exploration, the search yields a locally optimal solution. Moreover, containers not involved in W are not considered by the cost model, and a reasonable choice could be to compress them using order-unaware algorithms offering good compression ratios, e.g. `bzip2` [23]. Finally, note also that the choice of a suitable compression configuration is orthogonal with respect to the choosing of an optimal XML storage model [22]; we can combine both for an automatic storage-and-compression design.

4 Evaluating XML queries over compressed data

The XQueC query processor consists of a query parser, an optimizer, and a query evaluation engine. The set of physical operators used by the query evaluation engine can be divided in three classes:

– *data access operators*, retrieving information from the compressed storage structures;

– *regular data combination operators* (joins, outer joins, selections etc.);
– *compression and decompression operators*.

Among our data access operators, there are *ContScan* and *ContAccess*, which allow, respectively, to scan all (elementID, compressed value) pairs from a container, and to access only some of them, according to an interval search criteria. *StructureSummaryAccess* provide direct access to the identifiers of all elements reachable through a given path. *Parent* and *Child* allow to fetch the parent, respectively, the children (all children, or just those with a specific tag) for a given set of elements. Finally, *TextContent* pairs element IDs with all their immediate text children, retrieved from their respective containers. *TextContent* is implemented as a hash join pairing the element IDs with the content obtained from a *ContScan*.

Due to the storage model chosen in XQueC (Section 2.2), the *StructureSummaryAccess* operator provides the identifiers of the required elements *in the correct document order*. Furthermore, the *Parent* and *Child* operator preserve the order of the elements with respect to which they are applied. Also, if the *Child* operator returns more than one child for a given node, these children are returned in correct order. The order-preserving behavior allow us to perform many path computations through comparatively inexpensive 1-pass merge joins; furthermore, many simple queries can be answered without requiring a sort to re-constitute document order.

While these operators respect document order, *ContScan* and *ContAccess* respect data order, provides fast access to elements (and values) according to a given value search criteria. Also, as soon as predicates on container values are given in the query, it is often profitable to start query evaluation by scanning (and perhaps merge-joining) a few containers.

As an example of QEP, consider query Q9 from XMark:

```
FOR $p IN document("auction.xml")/site/people/person
LET $a :=
    FOR $t IN document("auction.xml")/site/
        closed_auctions/closed_auction
    LET $n :=
        FOR $t2 IN document("auction.xml")/site/
            regions/europe/item
        WHERE $t/itemref/@item = $t2/@id
        RETURN $t2
    WHERE $p/@id = $t/buyer/@person
    RETURN <item> $n/name/text() </item>
RETURN <person name=$p/name/text()> $a </person>
```

Figure 5 shows a possible XQueC execution plan for Q9 (this is indeed the plan used in the experiments). Based on this example, we make several remarks. First, note that we only decompress the necessary pieces of information (person name and item name), only at the very end of the query execution (the decompress operators shown in bold fonts). All the way down in the QEP, we were able to compute the three-ways join between persons, buyers, and items, using directly the compressed attributes person/@id, buyer/@person, and item_ref/@item. Second, due to the order of data obtained from *ContainerScans*, we are able to make extensive use of MergeJoins, without the need for sorting. Third, this plan mixes *Parent* and *Child* operators, alternating judiciously between top-down and bottom-up strategy, in order to minimize the number of tuples manipulated at any particular moment. This feature is made possible by the usage of a full set of algebraic evaluation choices, which XQueC has, but is not available to the XGrind or XPress query processors.

Document	*Size(MB)*	*Containers*	*Distinct tags*	*Tree nodes*
Shakespeare	15.0	39	22	65621
Baseball	16.8	41	46	27181
Washington-course	12.1	12	18	99729
XMark11	11.3	432	77	76726

Table 1. Data Sets used in the experiments (XMark11 is used in QETs measures.)

Finally, note that for instance in query Q9 also an *XMLSerialize* operator is employed in order to correctly construct the new XML which the query outputs. To this purpose, we recall that XML construction plays a minor role within the XML algebraic evaluation, and, being not crucial, it can be disregarded in the whole query execution time [24]. This has been confirmed by our experiments.

5 Implementation and experimental evaluation

XQueC is being implemented entirely in Java, using as back-end an embedded database, Berkeley DB [25]. We have performed some interesting comparative measures, that show that XQueC is a competitor of both query-aware compressors, and of early XQuery prototypes.

In the following, we want to illustrate both XQueC good compression ratios and query execution times. To this purpose, we have done two kinds of experiments:

Compression Factors. We have performed experiments on both synthetic data (XMark documents) and on real-life data sets (in particular, we considered the ones chosen in [5] for the purpose of cross-comparison with it).

Query Execution Times. We show how our system performs on some XML benchmark queries [8] and cross-compare them with the query execution times of optimized Galax [10], an open-source XQuery prototype.

All the experiments have been executed on a *DELL Latitude C820* laptop equipped with a 2,20GHz CPU and 512MB RAM.

Compression Factors. We have compared the obtained compression factors (defined as $1 - (cs/os)$), where cs and os are the sizes of the compressed and original documents, respectively) with the corresponding factors of XMill, XGrind and XPRESS. Figure 6 (left) shows the average compression factor obtained for a corpus of documents composed of *Shakespeare.xml*, *Washington-Course.xml* and *Baseball.xml*, whose main characteristics are shown in Table 1. Note that, on average, XQueC closely tracks XPRESS. It is interesting to notice that some limitations affect some of the XML compressors that we tested - for example, the documents decompressed by XPRESS have lost all their white spaces. Thus, the XQueC compression factor could be further improved if blanks were not considered.

Moreover, we have also tested the compression factors on different-sized XMark synthetic data sets (we considered documents ranging from 1MB to 25MB), generated by means of xmlgen [8]. As Figure 6 (right) shows, we have obtained again good compression factors w.r.t XPRESS and XMill.

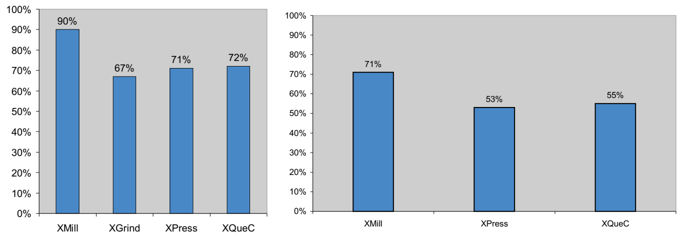


Fig. 6. Average CF for Shakespeare, WashingtonCourse and Baseball data sets (left); and for XMark synthetic data sets (right).

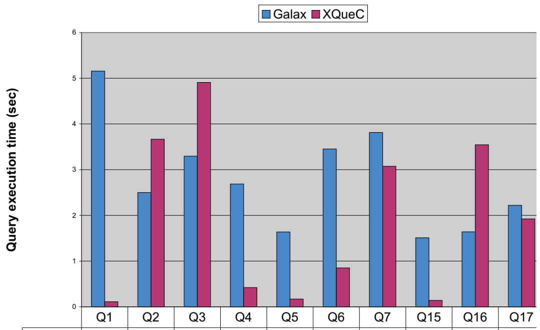
Note also that XGrind does not appear in these experiments. Indeed, due to repetitive crashes, we were not able to upload in the XGrind system (the version available through the site *http://sourceforge.net*) any XMark document except for one sized 100KB, whose compression factor however is very low and not representative of the system (precisely equal to 17.36%).

Query Execution Times. We have tested our system against the optimized version of Galax by running XMark queries and other queries. Due to space limits, we select here a set of significant XMark queries. Indeed, XMark queries left out stress language features, on which compression will likely have no significant impact whatsoever, e.g., support for functions, deep nesting etc. The reasons why we chose Galax is that it is open-source and has an optimizer. Note that the XQueC optimizer is not finalized yet (and was indeed not used in these measures), thus our results are only due to the compression ratio, data structures, and efficient execution engine.

Figure 7 shows the executions of XQueC queries on the document XMark11, sized 11.3MB. For the sake of better readability, in Figure 7, we have omitted Q9, and Q8. These queries measured in our system 2.133 sec. and 2.142 sec. respectively, whereas in Galax Q9 could not be measured on our machine [9] and Q8 took 126.33 sec. Note also that on Q2, Q3, Q16, the QET is a little worse than the Galax one, because in the current implementation we use simple unique IDs, given that our data model imposes a large number of parent-child joins. However, even with this limitation, we are still reasonably close to Galax, and we expect much better once XQueC will migrate to 3-valued IDs, as already started in the spirit of [26], [27], [28]. Most importantly, note that the previous XQueC QETs are to be intended as the times taken to both execute the queries in the compressed and decompress the obtained results. Thus, those measures show that there is no performance penalty in XQueC w.r.t. Galax due to compression. Thus, with comparable times w.r.t. an XQuery engine over uncompressed data, XQueC exhibits the advantage of compression.

As a general remark, note that our system is implemented in Java and can be made faster by using a native code compiler, which also we plan to plug in the immediate future.

[9] The same query has been tested on a more powerful machine in the paper [10] and results in a rather lengthy computation.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q15	Q16	Q17
Galax	5.156	2.5	3.295	2.687	1.637	3.453	3.814	1.51	1.64	2.22
XQueC	0.111	3.666	4.907	0.42	0.17	0.851	3.074	0.14	3.545	1.923

Fig. 7. Comparative execution times between us and Optimized Galax .

Finally, it is worth noting that comparison of XQueC with XGrind and XPress query times could not be done due to the fact that fully working versions of the latters are not publicly available. Nevertheless, that comparison would have been less meaningful, since those systems cover a limited fragment of XPath, and not full XQuery, as discussed in Section 1.2.

6 Conclusions and future work

We have presented XQueC, a compression-aware XQuery processor. We have shown that our system exhibits a good trade-off between compression factors over different XML data sets and query evaluation times on XMark queries. XQueC works on compressed XML documents, which can be a huge advantage when query results must be shipped around a network.

In the very near future, our system will be improved in several ways: by moving to three-valued IDs for XML elements, in the spirit of [26], [27], [28] and by incorporating further storage techniques that lead to additionally reduce the occupancy of structures. The implementation of an XQuery [29] optimizer for querying XML compressed data is ongoing. Moreover, we are testing the suitability of our system w.r.t. the full-text queries [30], which are being defined for the XQuery language at W3C. Another important extension we have devised is needed for uploading in our system larger documents than currently (e.g. SwissProt, measuring about 500MB). To this purpose, we plan to access the containers during the parsing phase directly on secondary storage rather than in memory.

References

1. Westmann, T., Kossmann, D., Helmer, S., Moerkotte, G.: The Implementation and Performance of Compressed Databases. ACM SIGMOD Record **29** (2000) 55–67
2. Chen, Z., Gehrke, J., Korn, F.: Query Optimization In Compressed Database Systems. In: Proc. of ACM SIGMOD. (2000)

3. Chen, Z., Seshadri, P.: An Algebraic Compression Framework for Query Results. In: Proc. of the ICDE Conf. (2000)
4. Tolani, P., Haritsa, J.: XGRIND: A query-friendly XML compressor. In: Proc. of the ICDE Conf. (2002)
5. Min, J.K., Park, M., Chung, C.: XPRESS: A queriable compression for XML data. In: Proc. of ACM SIGMOD. (2003)
6. Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: XQueC: Pushing XML Queries to Compressed XML Data (demo). Proc. of the VLDB Conf. (2003)
7. Liefke, H., Suciu, D.: XMILL: An efficient compressor for XML data. In: Proc. of ACM SIGMOD. (2000)
8. Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proc. of the VLDB Conf. (2002)
9. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML. In: Proc. of the VLDB Conf. (2003)
10. Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of the VLDB Conf. (2003)
11. Huffman, D.A.: A Method for Construction of Minimum-Redundancy Codes. In: Proc. of the IRE. (1952)
12. Antoshenkov, G.: Dictionary-Based Order-Preserving String Compression. VLDB Journal **6** (1997) 26–39
13. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing Relations and Indexes. In: Proc. of the ICDE Conf. (1998) 370–379
14. Poess, M., Potapov, D.: Data Compression in Oracle. In: Proc. of the VLDB Conf. (2003)
15. Moura, E.D., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and Flexible Word Searching on Compressed Text. ACM Transactions on Information Systems **18** (2000) 113–139
16. Witten, I.H.: Arithmetic Coding For Data Compression. Communications of ACM (1987)
17. Hu, T.C., Tucker, A.C.: Optimal Computer Search Trees And Variable-Length Alphabetical Codes. SIAM J. APPL. MATH **21** (1971) 514–532
18. Moffat, A., Zobel, J.: Coding for Compression in Full-Text Retrieval Systems. In: Proc. of the Data Compression Conference (DCC). (1992) 72–81
19. Antoshenkov, G., Lomet, D., Murray, J.: Order preserving string compression. In: Proc. of the ICDE Conf. (1996) 655–663
20. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice-Hall (1999)
21. Amer-Yahia, S.: Storage Techniques and Mapping Schemas for XML. SIGMOD Record (2003)
22. Bohannon, P., Freire, J., Roy, P., Simeon, J.: From XML Schema to Relations: A Cost-based Approach to XML Storage. In: Proc. of the ICDE Conf. (2002)
23. Website: The bzip2 and libbzip2 Official Home Page (2002) http://sources.redhat.com/bzip2/.
24. Shanmugasundaram, J., Shekita, E., Barr, R., Carey, M., Lindsay, B., Pirahesh, H., Reinwald, B.: Efficiently Publishing Relational Data as XML Documents. In: Proc. of the VLDB Conf. (2000)
25. Website: Berkeley DB Data Store (2003) http://www.sleepycat.com/pro-ducts/data.shtml.
26. Paparizos, S., Al-Khalifa, S., Chapman, A., Jagadish, H.V., Lakshmanan, L.V.S., Nierman, A., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: TIMBER:A Native System for Querying XML. In: Proc. of ACM SIGMOD. (2003) 672
27. T.Grust: Accelerating XPath location steps. In: Proc. of ACM SIGMOD. (2002) 109–120
28. Srivastava, D., Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Wu, Y.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proc. of the ICDE Conf. (2002)
29. Website: The XML Query Language (2003) http://www.w3.org/XML/Query.
30. Website: XQuery and XPath Full-text Use Cases (2003) http://www.w3.org/TR/xmlquery-full-text-use-cases.