# The XQueC Project: Compressing and Querying XML

Andrei Arion *
SCORT SA, France
Angela Bonifati
ICAR CNR, Italy
Ioana Manolescu
INRIA Saclay–Île-de-France, France
Andrea Pugliese
DEIS – University of Calabria, Italy

**Abstract**

We outline in this paper the main contributions of the XQueC project. XQueC, namely XQuery processor and Compressor, is the first compression tool to seamlessly allow XQuery queries in the compressed domain. It includes a set of data structures, that basically shred the XML document into suitable chunks linked to each other, thus disagreeing with the 'homomorphic' principle so far adopted in previous XML compressors. According to this principle, the compressed document is homomorphic to the original document. Moreover, in order to avoid the time consumption due to compressing and decompressing intermediate query results, XQueC applies 'lazy' decompression by issuing the queries directly in the compressed domain.

**Terms**: XML databases, XML compression
**Keywords**: XML compression, Data structures, XQuery querying

## 1 Introduction

An increasing amount of data on the Web is now available as XML, either being directly created in this format, or exported to XML from other formats. XML documents typically exhibit a high degree of redundancy, due to the repetition of element tags, and an expensive encoding of the textual content. As a consequence, exporting data from proprietary formats to XML typically increases its volume significantly. For example, [LS00] shows that specific format data, such as Weblog data [APA04] and SwissProt data [UWX04], once XML-ized grow by about 40%.

The redundancy often present in XML data provides opportunities for compression. In some applications (e.g., data archiving), XML documents can be compressed with a general-purpose algorithm (e.g., GZIP), kept compressed, and rarely decompressed. However, other

*The work has been done while the author was affiliated with INRIA Saclay (France).

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1691

applications, in particular those frequently querying compressed XML documents, cannot afford to fully decompress the entire document during query evaluation, as the penalty to query performance would be prohibitive. Instead, decompression must be carefully applied on the minimal amount of data needed for each query.

With this in mind we have designed XQueC, a full-fledged data management system for compressed XML data. XQueC is equipped with a compression-compliant storage model for XML data, which allows many storage options for the query processor. The XQueC storage model leverages a proper data fragmentation strategy, which allows the identification of the units of compression (*granules*) for the query processor. These units are also manipulated at the physical level by the storage backend.

XQueC's data fragmentation strategy is based on the idea of separating structure and content within an XML document. It often happens that data nodes found under the same path exhibit similar and related content. Therefore, it makes sense to group all such values into a single *container* and to decide upon a compression algorithm *once per container*. The idea of using data containers has been borrowed from the XMill project [LS00]. However, whereas XMill compressed and handled a container as a whole, in XQueC each container item (corresponding to a data node) is individually compressed and accessible. The containers are key to achieving good compression as the PCDATA of a document affects the final document compression ratio more than the tree of tags (which is typically only 20%-30% of the overall compressed document size).

XQueC's fragmented storage model supports fine-grained access to individual data items, providing the basis for diverse efficient query evaluation strategies in the compressed domain. It is also transparent enough to process complex XML queries. By contrast, other existing XML queryable compressors exploit coarse-grained compressed formats, thus only allowing a single top-down evaluation strategy.

In the XQueC storage model, containers are further aggregated into *groups*, which allow their data commonalities to be exploited, thus allowing both compression and querying to be improved. In addition to the space usage of compressed containers itself, there are several other factors that impact the final compression ratio and the query performance. Consider for instance two containers: if they belong to the same group, they will share the same source model, i.e., the support structure used by the algorithm (e.g., a tree in the case of the Huffman algorithm); if instead they belong to separate groups, they have separate source models, thus always requiring decompression in order to compare their values. Therefore, the grouping method impacts both the containers space usage and the decompression times.

A proper choice of how to group containers should ensure that containers belonging to the same group also appear together in query predicates. Indeed, it is always preferable to perform the evaluation of a predicate within the compressed domain; this can be done if the containers involved in the predicate belong to the same group and are compressed with an algorithm supporting that predicate in the compressed domain. Information about predicates can be inferred by looking at available query workloads. Moreover, different compression algorithms may support different kinds of predicates in the compressed domain: for instance, the Huffman algorithm [Huf52] allows the evaluation of equality predicates, whereas

the ALM algorithm [Ant97] supports both equality and inequality predicates. XQueC addresses these issues by employing a *cost model* and applying a suitable blend of heuristics to make the final choice.

Since XQueC is capable of carefully balancing different compression performance aspects, it can be considered as a full-fledged compressed XML database, rather than a simple compression tool. In summary, XQueC is the first queryable XML database management system capable of:

- exploiting a storage model based on a fragmentation strategy that supports complex XML queries and enables efficient query processing;
- compressing XML data and querying it as much as possible in the compressed domain;
- making a cost-based choice of the compression granules and corresponding compression algorithms, possibly based on a given query workload.

We demonstrate the utility of XQueC by means of a wide set of experimental results on a variety of XML datasets and by comparing it with available competitor systems.

The remainder of the paper is organized as follows. Section 2 discusses the related literature and presents a summary of the differences among XQueC and the available XML compression tools. Section 3 illustrates the XQueC storage model.

# 2 Related Work

Compression has long been recognized as a useful means to improve the performance of relational databases [CGK00, WKHM00, AYJ00]. However, the results obtained in the relational domain are only partially applicable to XML. We examine in this section the existing literature on compression as studied for relational databases, explaining to what extent it might or might not be applicable to XML, and then survey the existing tools for compression and querying of XML data [NLC06].

## 2.1 Compression in relational databases

First of all, let us note that the interest in compressing relational data has focused primarily on numerical attributes. String attributes, which are less frequent in relational schemas, have received much less attention. In contrast, string content is obviously critical in the XML context. For example, within the TPC-H [Tra99] benchmark schema, only 26 of 61 attributes are strings, whereas, within the XMark [SWK$^+$02] benchmark for XML databases, 29 out of the 40 possible element content (leaf) nodes represent string values.

Studies of compression for relational databases include [CGK00, GRS98, Gra93, Gre99, WKHM00]. The focus of these works has been on *(i)* effectively compressing terabytes of data, and *(ii)* finding the best compression granularity (field-, block-, tuple-, and file-level) from a query performance perspective. [WKHM00] discusses light-weight relational compression techniques oriented to field-level compression, while [Gre99] uses both record-level and field-level encodings. Unfortunately, field-level and record-level compression do

not translate directly to the XML context. [GRS98] proposes an encoding, called *FOR* (frame of reference), to compress numeric fact tables fields, that elegantly blends page-at-a-time and tuple-at-a-time decompression. Again, their results clearly do not translate to XML.

These papers have also studied the impact of compression on the query processor and the query optimizer. While Goldstein et al. [GRS98] applies compression to index structures, such as B-trees and R-trees, to reduce their space usage, [WKHM00] discusses how to modify the relational query processor, the storage manager, and the query optimizer in presence of field-level compression. [CGK00] focuses on query optimization for compressed relational databases, by introducing *transient* decompression, i.e., intermediary results are decompressed (e.g., in order to execute a join in the compressed domain), then re-compressed for the rest of the execution. As XQueC does for XML data, both [CGK00] and [WKHM00] address the problem of incorporating compression within databases in the presence of possibly poor decompression performance, which may outweigh the savings due to fewer disk accesses.

A novel lossy semantic compression algorithm oriented toward relational data mining applications is presented in [JNOT04]. Finally, compression in a data warehouse setting has been applied in commercial DBMS products such as Oracle [PP03]. The recent advent of the concept of *Web mart* (Web-scale structured data warehousing, currently pursued by Microsoft, IBM and Sun) leads to the possibility that the interest of compression for data warehouses will shift from the relational model to XML in the near future.

### 2.2 Non-queryable compressors for XML databases

XMill [LS00] is a pioneering system for efficiently compressing XML documents. It is based on the principle of separately compressing the values and the document tags. Values are assigned to containers in a default way (one container for each distinct element name) or, alternatively, in a user-driven way. In order to achieve both maximum compression rate and time, XMill may use a customized semantic compressor, and the obtained result may be re-compressed with either GZIP or BZIP2 [BZI02].

XMLZIP [XML99] compresses an XML document by clustering subtrees from the root to a certain depth. This does not allow the exploitation of redundancies that may appear below this fixed level, and hence some compression opportunities are lost.

Another query-oblivious compressor which exploits the XML hierarchical structure is XMLPPM [Che01]. It implements ESAX, an extended SAX parser, which allows the online processing of documents. XMLPPM does not require user input, and can achieve better compression than XMill in the default mode. However, it still represents a relatively slow compressor when compared to XMill. A variant of XMLPPM that looks at the DTD to improve compression has been recently presented [Che05].

The three compressors above focus on achieving the maximum compression for XML data and are not transparent to queries.

### 2.3 Queryable compressors for XML databases

Our work is most directly comparable with queryable XML compression systems.

The XGrind system [TH02] compresses XML by using a *homomorphic* encoding: an XGrind-compressed XML document is still an XML document, whose tags have been encoded by integers and whose textual content has been compressed using the Huffman (Dictionary, alternatively) algorithm. The XGrind query processor is an extended SAX parser, which can handle exact-match and prefix-match queries in the compressed domain. Most importantly, XGrind only allows a top-down query evaluation strategy, which may not always be desirable. XGrind covers a limited set of XPath queries, allowing only child and attribute axes. It cannot handle many query operations, such as inequality selections in the compressed domain, joins, aggregations, nested queries, and XML node construction. Such operations occur in many XML query scenarios (e.g., all but the first two of the 20 XMark [SWK$^+$02] benchmark queries).

XPRESS [MPC03] encodes whole paths into floating point numbers, and, like XGrind, compresses textual (numeric, resp.) leaves using the Huffman (Difference or Dictionary, alternatively) encoding. The novelty of XPRESS lies in its *reverse arithmetic* path encoding scheme, which encodes each path as an interval of real numbers between 0 and 1. Queries supported in the compressed domain amount to exact/prefix queries and range queries with numerical values. Range queries with strings require full decompression. Also, the navigation strategy is still top-down as the document structure is maintained by homomorphism. The fragment of XPath supported is more powerful than the one in XGrind, as it also allows descendant axes. A recent extension of XPRESS [MPC06] replaces the Huffman encoding with the Arithmetic encoding, thus preserving the order information among data values. It also handles simple updates on XML data, such as insertions of new XML fragments or deletions of existing ones. The compressed engine recomputes the statistics for the newly added (or removed) content and only decompresses the portions of the document affected by the changes.

In [BGK03] compression is applied to the structure of an XML document by using a bisimulation relationship, whereas leaf textual nodes are left uncompressed. This compressed structure preserves enough information to directly support *Core XPath* [MS02], a rich subset of XPath. A more recent paper [BLM05] proposes a similar compact representation for XML binary trees, based on sharing common subtrees. However, both systems cannot be directly compared with XQueC, because they are memory-based, and do not produce a persistent compressed image of the data instance.

XQZip [CN04] uses a structure index tree (SIT) that tends to merge subtrees containing the exact same set of paths. It applies GZIP compression to value blocks, which entails decompressing entire blocks during query evaluation. The blocks have a predefined length, empirically set at $1,000$ records each. At query processing time, XQZip tries to determine the minimum number of blocks to be decompressed. The queries addressable by XQZip belong to an extended version of XPath, enriched with union and the grouping operator in the return step.

Finally, XCQ [NLWL06] uses DTDs to perform compression and subsequent querying

| *System* | *Struct./Text Compression* | *Homomorph.* | *Predicates* | *Language* | *Evaluation strategies* | *Compression granules* |
|---|---|---|---|---|---|---|
| XGrind | Binary/ Huffman+Dict. | Yes | =, prefix | XPath subset | Top-down | Value/tag |
| XPRESS | RAE/ Huffman(Arith.)+ Dictionary+Diff. | Yes | =,<, prefix | XPath subset++ | Top-down | Value/path |
| [BGK03] | Bisimulation/ — | No | — | Core XPath | Top-down bottom-up | — |
| XQZip | SIT/ GZip | No | — | XPath 1.0++ | Multiple | Block (set of records) |
| XCQ | PPG/ GZip | No | — | XPath 1.0 + aggr. | Multiple | Block (set of records) |
| XQueC | Binary/ cost-driven | No | =, <, prefix | XQuery subset | Multiple | Container item/tag |

Table 1: Comparative analysis of queryable XML compressors.

of XML documents. Partitioned path-based grouping (PPG) data streams are obtained for each DTD path, and then compressed into a number of data blocks, which are input to GZIP afterwards. Similarly to XQZip, the block size has to be carefully determined in order to achieve good performance.

Table 1 reports the major differences among the discussed systems. XQueC realizes a cost-driven compression, and a random-access query evaluation strategy, as opposed to XPRESS, XGrind and XQZip. This is what makes XQueC the first compressed XML database, rather than an XML compression tool. Besides guaranteeing that queries are processed as much as possible in the compressed domain, XQueC also supports a more expressive language fragment. Finally, the level of granularity XQueC considers is the smallest possible, i.e., a container item or a tag, which can be thus randomly accessed during querying. This is similar to XGrind, and in contrast to XQZip/XCQ, which rely on block-level granules, and to XPRESS, which has both value-level and path-level granules.

# 3 Storing and querying compressed XML data

In this section, we describe XQueC's storage model for compressed XML data. We outline XQueC's overall architecture in Subsection 3.1. XQueC's query processing model is briefly described in Subsection 3.2. This provides the groundwork for discussing the trade-off between compact storage and efficient querying (Subsection 3.3).

## 3.1 XQueC storage structures and architecture

XQueC splits an XML document into three data structures, depicted in Fig. 1 for an XMark sample: the *structure tree*, the *containers* and the *structure summary*. Besides providing a description of each data structure, in the following we also discuss its space usage in order to give an insight on the impact of each storage structure on the final document's compression ratio.

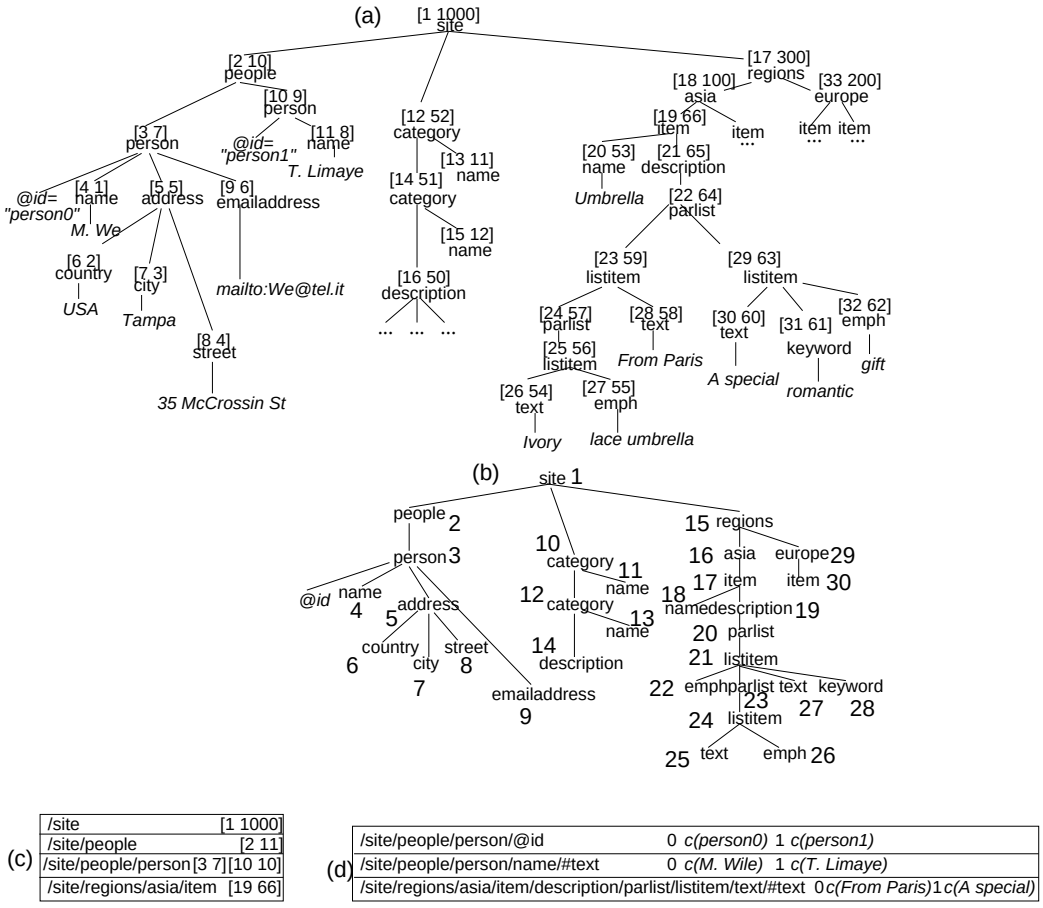


Figure 1: XQueC storage structures: (a) sample XMark document, (b) structure summary, (c) ID sequences and (d) containers.

Across all the structures, XQueC encodes element and attribute names using a simple binary encoding. The structure tree is encoded as a set of ID sequences, each associated with a different root-to-node path in the tree. Figure 1(c) depicts the sequences resulting from the paths /site, /site/people, /site/people/person, and /site/regions/asia/item in the sample document. To encode the IDs in all its storage structures, XQueC uses conventional *structural identifiers* consisting of triples [pre, post, depth] as in [AKJP+02, HBG+03, PAKC+03, Gru02]. The pre (post) number reflects the ordinal position of the element within the document, when traversing it in preorder (postorder). The depth number reflects the depth of the element in the XML tree. This node identification scheme allows the direct inference of structural relationship between two nodes using only their identifiers. Note that the depth field can be omitted, since in our storage structures, the structural identifiers are already clustered by their path. Thus, the sequences in Fig. 1(c) actually use only a 2-tuple [pre, post] to encode each structural ID. This means that for a document having $N$ elements, each [pre, post] ID is encoded using $2 * \lceil log_2(N) \rceil$ bits, thus the space usage of the set of ID sequences is

$$cs_{seq} = 2 * N * \lceil log_2(N) \rceil. \tag{1}$$

Similarly, the containers store together all data values found under the same root-to-leaf path in the document. A container is realized as a sequence of records, each consisting of a compressed value, and a number representing the position of its parent in the corresponding ID sequence of the tree structure (see Fig. 1(d), where *c(s)* denotes the compressed version of string *s* [1]). We write $size(c_i)$ for the size in bits of the $i$-th compressed value in container $c$ and $seq_c$ for the ID sequence of its parent. Hence, the space usage of the compressed containers is

$$cs_{cont} = \sum_{c} \Big( |c| * \lceil log_2(|seq_c|) \rceil + \sum_{i=1,\ldots,|c|} size(c_i) \Big). \tag{2}$$

Finally, the storage model includes a *structure summary*, i.e., an access support structure storing all the distinct paths in the document. The structure summary of an XML document $d$ is a tree whose nodes uniquely represent the paths in $d$, that is, for each distinct path $p$ in $d$, the summary has exactly one node on path $p$. For a textual node under path $p$, the summary has a node labeled $/p/\#$text, whereas for an attribute node $a$ under path $p$, the summary has a node labeled $/p/@a$. This establishes a bijection between paths in an XML document and nodes in the structure summary. Note also that each leaf node in the structure summary uniquely corresponds to a container of compressed values. Fig. 1(b) depicts the structure summary for the sample document. The space usage of a summary $SS$ is:

$$cs_{aux} = \sum_{n \in SS} \Big( |tag(n)| + log_2(|SS|) \Big). \tag{3}$$

[1] When type information is not known a priori, XQueC applies a simple type inference algorithm that attempts to classify the values on each path into simple primitive types.

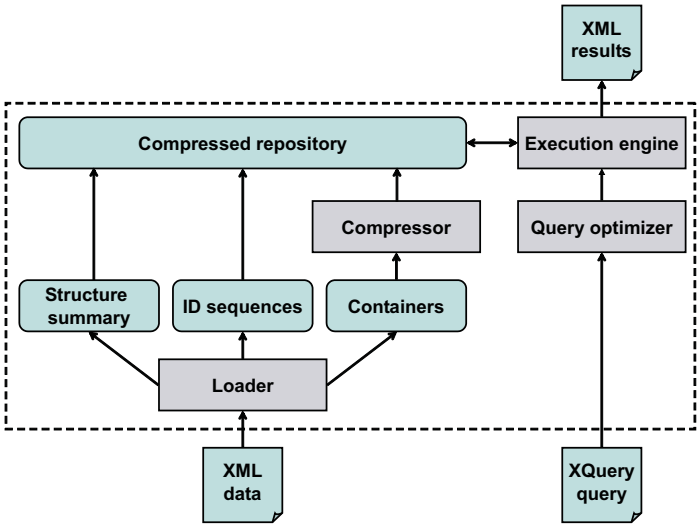


Figure 2: Architecture of the XQueC prototype.

where the first term represents the space needed for the storage of each node's tag and the second term accounts for its incoming edge. The summary is typically very small (see the details in [ABMP07]), thus it does not significantly impact data compression.

Overall, the compressed document size is thus $cs = cs_{seq} + cs_{cont} + cs_{aux}$, and the resulting compression factor is $cf = 1 - cs/os$, where $os$ is the original document size.

Fig. 2 outline XQueC's architecture. The loader decomposes the XML document into ID sequences and containers, and builds the structure summary. The compressor partitions the data containers and decides which algorithm to apply (cfr. [ABMP07]). This phase produces a set of compressed containers. The repository stores the storage structures and provides data access methods and a set of compression functions working at runtime on constant values appearing in the query. Finally, the query processor includes a query optimizer and an execution engine providing the physical data access operators.

### 3.2 Processing XML queries in XQueC

The XQuery subset $\mathcal{Q}$ supported by XQueC is characterized as follows.
**(1)** $\text{XPath}^{\{/,//,*,[\,]\}} \subset \mathcal{Q}$, that is, any Core XPath belongs to $\mathcal{Q}$. When such XPath expressions have as suffix a call to the function *text()*, they return the text value of the nodes they are applied on. Navigation branches enclosed in square brackets may include complex paths and comparisons between a node and a constant $c$. Predicates connecting two nodes are not allowed; they may be expressed in XQuery syntax, as explained next. **(2)** Let $\$x$ be a variable bound in the query context [XQU04] to a list of XML nodes, and $p$ be a Core XPath expression. Then, $\$x\,p$ belongs to $\mathcal{Q}$, and represents the path expression $p$ applied with $\$x$'s bindings list as initial context list. For instance, $\$x/a[c]$ returns the $a$ children of $\$x$ bindings having a $c$ child. We denote the set of expressions (1) and (2) above as $\mathcal{P}$, the

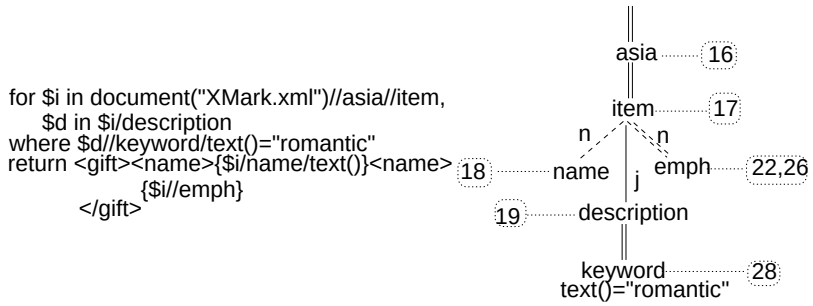


Figure 3: Sample XQuery expression, and its corresponding path-annotated query pattern.

set of path expressions. **(3)** For any two expressions $e_1$ and $e_2 \in \mathcal{Q}$, their concatenation, denoted $e_1, e_2$, also belongs to $\mathcal{Q}$. **(4)** If $t$ is a tag and $e \in \mathcal{Q}$, element constructors of the form $\langle t \rangle \{e\} \langle /t \rangle$ belong to $\mathcal{Q}$. **(5)** All expressions of the following form belong to $\mathcal{Q}$:

$$\boxed{xq} \quad \begin{array}{l} \text{for } \$x_1 \text{ in } p_1, \$x_2 \text{ in } p_2, \ldots, \$x_k \text{ in } p_k \\ \text{where } p_{k+1}\ \theta_1\ p_{k+2} \text{ and } \ldots \text{ and } p_{m-1}\ \theta_l\ p_m \\ \text{return } q(x_1, x_2, \ldots, x_k) \end{array}$$

where $p_1, p_2, \ldots, p_k, p_{k+1}, \ldots, p_m \in \mathcal{P}$, any $p_i$ starts either from the root of some document $d$, or from a variable $x_l$ introduced in the query before $p_i$, $\theta_1, \ldots, \theta_l$ are some comparators, and $q(x_1, \ldots, x_k) \in \mathcal{Q}$. A return clause may contain other for-where-return queries, nested and/or concatenated and/or grouped inside constructed elements.

XQueC's optimizer compiles a query $q \in \mathcal{Q}$ into an executable plan in several steps.

First, a set of query patterns, capturing $q$'s path expressions and the relationships among them, are extracted from $q$. Figure 3 shows a query and its corresponding pattern, in which child (resp. descendant) pattern edges are shown by simple (resp. double) lines, and optional edges (allowing matches for the descendant node to be missing) are shown in dashed lines. Finally, $n$ markers identify nested edges: matches of the lower node should be *nested* under the upper node matches. For instance, all name and emph matches should be output together for a given $i and $d match. The full pattern extraction algorithm, which is beyond the scope of this paper, is described in [ABM+06].

Based on the structure summary, XQueC analyzes each query pattern, associating to each pattern node all paths (from the XML document) where bindings for this pattern node may be found. In Figure 3, the numbers identifying the summary paths (recall Figure 1) associated to each node are shown in dotted circles next to the node. This analysis follows the original Dataguide usage for optimization [GW97].

The optimizer then builds a data access plan for each pattern node. If the query requires the text value of the pattern node, such as $name in Figure 3, the access plan reads the contents of containers corresponding to those paths. Otherwise, the access plan reads the ID sequences for those paths. In both cases, unions are built whenever a pattern node has more than one associated path, as was the case, for instance, with the emph in Figure 3.

Data access plans corresponding to pattern nodes are combined by structural join operators [AKJP+02] reflecting the semantics of pattern edges. We use structural outerjoins

for optional edges, as proposed in [CJLP03]. Structural joins followed by grouping are employed for nested pattern edges.

To compensate for XQueC's highly partitioned storage, the optimizer must produce plans that *reconstruct* the XML elements which the query needs to output entirely, such as emph in Figure 3. One alternative is to combine all the necessary containers and ID sequences via structural joins. Another alternative is based on a pipelined, memory-efficient operator, which we have studied in [ABMP06].

Finally, XQueC's optimizer adds decompression operators, to decompress those values that must be returned (uncompressed) in the query results.

### 3.3 Trade-offs between compact storage and efficient processing

XQueC aims at providing efficient query processing techniques typical of XML databases together with the advantages of XML compression. These two goals clearly conflict. For instance, compressing blocks of several values at a time (instead of compressing each value individually, as XQueC does) may improve the compression factor, but would reduce the query engine's ability to perform very selective data access.

The desired XML database features which we targeted in XQueC are: selective data access; scalable query execution operators; and low memory needs during query processing. Our goals for XML compression in XQueC were: to reduce space usage, and to decompress lazily. XQueC's design is the result of mediating between these desiderata, as outlined below.

Path partitioning provides for selective data access, more so than the tag-partitioning structural ID indexing used in [JAKC+02, FHK+02, HBG+03]. Node partitioning schemes more aggressive than path partitioning can be envisioned [BGK03], but they may lead lead to excessive fragmentation. Structure Index Trees (SIT) [CN04] also lead to partitioning nodes more than in XQueC, since two nodes are in the same group if they have the same incoming path *and the same set of outgoing paths*. For instance, on the XMark document of Figure 1(a), the two person elements would be in separate groups, since one has an address child while the other does not. In the presence of optional elements, the SIT may thus get very large.

Compressing each value individually enables both selective data access and lazy decompression. The separation between ID sequences and containers helps selective data access, since the processor does not have to access XML node *values* (voluminous even after compression) when it only needs to access (part of) the tree structure. For instance, for four out of the six pattern nodes in Figure 3, only ID sequences will be read. By the same argument, this separation also reduces the processor's memory needs.

To enable scalable query processing techniques in XQueC, we introduced structural identifiers for every node. The space occupied by the identifiers is the price to pay for the benefits of structural join algorithms that run in linear time and require low memory [AKJP+02]. Observe that homomorphic compressors such as XGrind and XPRESS, lacking a store, do not have direct access to given parts of the document. In such settings, there will always be "unlucky" queries whose processing requires a full traversal of the compressed document,

even if they only retrieve a small amount of data. Selective data access methods ensures that XQueC does not suffer from such problems, given that:

- each compressed value can be accessed directly;
- IDs from each document path can be accessed directly (and in the order favorable for further processing).

Path partitioning reduces IDs space usage by not storing the depth ID field; moreover, we only store the post-order number in the ID sequences (not in containers).

To store XML documents in a compact manner, XQueC cannot afford to complement ID sequences with a full persistent tree, as done in [JAKC+02, FHK+02, HBG+03], which (in the absence of value compression) report a disk footprint four times the size of the document. Thus, while ID sets are used as indices in [MS99, GW97], in XQueC they actually are the storage.

XQueC's elaborate choice of the best compression algorithm to use for each container is important for reducing storage size, but also for lazy decompression. Details can be found in [ABMP07].

# 4 Conclusions

In this paper, we have presented the XQueC project. XQueC allows to store compressed XML data and issue XQuery queries in the compressed domain. Among its main contributions, we have discussed the storage model, which includes a set of suitable data structures that allow to create accessible and efficient data containers. The system is also equipped with a cost model, in order to decide the compression techniques and the compression granules needed for compressing and querying the data. For a detailed description of such cost model and the experimental assessment of our system, we point the reader to [ABMP07].

# References


[ABM+06] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *Proceedings of the International Conference on Flexible Query Answering Systems*, pages 13–25, 2006.

[ABMP06] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. In *Proceedings of the International World Wide Web Conference*, pages 1077–1078, 2006.

[ABMP07] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. XQueC: A query-conscious compressed XML database. *ACM Trans. Internet Techn.*, 7(2), 2007.

[AKJP+02] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 141–152, San Jose, CA, USA, March 2002. IEEE.

[Ant97] Gennady Antoshenkov. Dictionary-Based Order-Preserving String Compression. *VLDB Journal*, 6(1):26–39, 1997.

[APA04] Apache custom log format. http://www.apache.org/docs/mod/mod_log_config.html, 2004.

[AYJ00] S. Amer-Yahia and T. Johnson. Optimizing Queries on Compressed Bitmaps. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 329–338, Cairo, Egypt, 2000. ACM.

[BGK03] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML . In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 141–152, Berlin, Germany, 2003. Morgan Kaufmann.

[BLM05] G. Busatto, M. Lohrey, and S. Maneth. Efficient Memory Representation of XML Documents. pages 199–216, Trondheim, Norway, 2005.

[BZI02] The bzip2 and libbzip2 Official Home Page, 2002. http://sources.redhat.com/bzip2/.

[CGK00] Z. Chen, J. Gehrke, and F. Korn. Query Optimization In Compressed Database Systems. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 271–282, Dallas, TX, USA, 2000. ACM.

[Che01] James Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *Data Compression Conference*, pages 163–172, Snowbird, Utah, USA, 2001. IEEE Computer Society.

[Che05] James Cheney. An Empirical Evaluation of Simple DTD-Conscious Compression Techniques. In *WebDB*, pages 43–48, 2005.

[CJLP03] Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 237–248, Berlin, Germany, 2003. Morgan Kaufmann.

[CN04] J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. In *Proceedings of the International Conference on Extending Database Technologies*, pages 219–236, Heraklion, Greece, 2004.

[FHK+02] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The Very Large Databases Journal*, 11(4):292–314, 2002.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Gre99] R. Greer. Daytona and the fourth-generation language Cymbal. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 525–526, Philadelphia, PA, USA, 1999. ACM.

[GRS98] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 370–379, Orlando, FL, USA, 1998. IEEE.

[Gru02] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 109–120, Madison, WI, USA, 2002. ACM.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, 1997. Morgan Kaufman.

[HBG+03] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A.N. Rao, F. Tian, S. Viglas, Y. Wang, J.F. Naughton, and D.J. DeWitt. Mixed Mode XML Query Processing. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 225–236, Berlin, Germany, 2003. Morgan Kaufmann.

[Huf52] D. A. Huffman. A Method for Construction of Minimum-Redundancy Codes. In *Proc. of the IRE*, pages 1098–1101, 1952.

[JAKC+02] H .V. Jagadish, S. Al-Khalifa, A. Chapman, L. V.S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, , and C. Yu. Timber: a native XML database. *The Very Large Databases Journal*, 11(4):274–291, 2002.

[JNOT04] H. V. Jagadish, R.T. Ng, B. Chin Ooi, and A. K. H. Tung. ItCompress: An Iterative Semantic Compression Algorithm. In *Proceedings of the International Conference on Data Engineering*, pages 646–658, Boston, MA, USA, 2004. IEEE Computer Society.

[LS00] H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, TX, USA, 2000. ACM.

[MPC03] J. Ki Min, M. Park, and C. Chung. XPRESS: A Queriable Compression for XML Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 122–133, San Diego, CA, USA, 2003. ACM.

[MPC06] J. Ki Min, M. Park, and C. Chung. A Compressor for Effective Archiving, Retrieval, and Update of XML Documents. *ACM Transactions On Internet Technology*, 6(3), 2006.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 277–295, 1999.

[MS02] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Conference on the Principles of Database Systems*, pages 65–76, 2002.

[NLC06] W. Ng, Y. W. Lam, and J. Cheng. Comparative Analysis of XML Compression Technologies. *World Wide Web Journal*, 9(1):5–33, 2006.

[NLWL06] W. Ng, Y. W. Lam, P. Wood, and M. Levene. XCQ: A Queriable XML Compression System (to appear). *International Journal of Knowledge and Information Systems*, 2006.

[PAKC+03] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, D. Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER:A Native System for Querying XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 672, San Diego, CA, USA, 2003. ACM.

[PP03] M. Poess and D. Potapov. Data Compression in Oracle. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 937–947, Berlin, Germany, 2003. Morgan Kaufmann.

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 974–985, Hong Kong, China, 2002. Morgan Kaufmann.

[TH02] P. Tolani and J. Haritsa. XGRIND: A Query-friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering*, pages 225–235, San Jose, CA, USA, 2002. IEEE.

[Tra99] Transaction processing performance council. TPC-H Benchmark Database., 1999. http://www.tcp.org.

[UWX04] University of Washington's XML repository. Available at www.cs.washington.edu/research/xmldatasets, 2004.

[WKHM00] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.

[XML99] XMLZip XML compressor. Available at http://www.xmls.com/products/xmlzip/xmlzip.html, 1999.

[XQU04] The XML Query Language, 2004. http://www.w3.org/XML/Query.