XQueC: A Query-Conscious Compressed XML Database

ANDREI ARION
INRIA Futurs—LRI, PCRI, France
ANGELA BONIFATI
ICAR CNR, Italy
IOANA MANOLESCU
INRIA Futurs—LRI, PCRI, France
and
ANDREA PUGLIESE
DEIS—University of Calabria, Italy

XML compression has gained prominence recently because it counters the disadvantage of the verbose representation XML gives to data. In many applications, such as data exchange and data archiving, entirely compressing and decompressing a document is acceptable. In other applications, where queries must be run over compressed documents, compression may not be beneficial since the performance penalty in running the query processor over compressed data outweighs the data compression benefits. While balancing the interests of compression and query processing has received significant attention in the domain of relational databases, these results do not immediately translate to XML data.

In this article, we address the problem of embedding compression into XML databases without degrading query performance. Since the setting is rather different from relational databases, the choice of compression granularity and compression algorithms must be revisited. Query execution in the compressed domain must also be rethought in the framework of XML query processing due to the richer structure of XML data. Indeed, a proper storage design for the compressed data plays a crucial role here.

The *XQ*ue*C* system (*XQ*uery Processor and *C*ompressor) covers a wide set of XQuery queries in the compressed domain and relies on a workload-based cost model to perform the choices of the compression granules and of their corresponding compression algorithms. As a consequence, XQueC provides efficient query processing on compressed XML data. An extensive experimental

A preliminary version of this paper appeared in the Proceedings of the 2004 International Conference on Extending DataBase Technology, pp. 200–218.
Authors' addresses: A. Arion and I. Manolescu, INRIA Futurs, Parc Club Orsay-Universite, 4 rue Jean Monod, 91893 Orsay Cedex, France; email: {firstname.lastname}@inria.fr; A. Bonifati, Icar CNR, Via P. Bucci 41/C, 87036 Rende (CS), Italy; email: bonifati@icar.cnr.it; A. Pugliese, DEIS—University of Calabria, Via P. Bucci 41/C, 87036 Rende(CS), Italy; email: apugliese@deis.unical.it.
 DOI 10.1145/1239971.1239974 http://doi.acm.org/10.1145/1239971.1239974

assessment is presented, showing the effectiveness of the cost model, the compression ratios, and the query execution times.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing, textual databases*; E.4 [**Coding and Information Theory**]: *Data compaction and compression*

General Terms: Algorithms, Design

Additional Key Words and Phrases: XML databases, XML compression, XML data management, XQuery

ACM Reference Format:

Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. 2007. XQueC: A query-conscious compressed XML database. ACM Trans. Intern. Tech. 7, 2, Article 10 (May 2007), 35 pages. DOI = 10.1145/1239971.1239974 http://doi.acm.org/10.1145/1239971.1239974

1. INTRODUCTION

An increasing amount of data on the Web is now available as XML, either 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 Liefke and Suciu [2000] show that specific format data, such as Weblog data [APA 2004] and SwissProt data [UWXML 2004], 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 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 [Liefke and Suciu 2000]. 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, that is, 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 [Huffman 1952] allows the evaluation of equality predicates, whereas the ALM algorithm [Antoshenkov 1997] 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 article 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. Section 4 presents the compression principles of XQueC and the cost model that makes the compression choices targeted to data and queries. Section 5 presents an extensive experimental study that probes both XQueC compression and querying capabilities. Section 6 concludes the article and discusses the future directions of our work.

2. RELATED WORK

Compression has long been recognized as a useful means to improve the performance of relational databases [Chen et al. 2000; Westmann et al. 2000; Amer-Yahia and Johnson 2000]. 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 [Ng et al. 2006].

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 [Transaction processing performance council 1999] benchmark schema, only 26 of 61 attributes are strings whereas within the XMark [Schmidt et al. 2002] benchmark for XML databases, 29 out of the 40 possible element content (leaf) nodes represent string values.

Studies of compression for relational databases include Chen et al. [2000], Goldstein et al. [1998], Graefe [1993], Greer [1999], and Westmann et al. [2000]. 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. Westmann et al. [2000] discusses lightweight relational compression techniques oriented to field-level compression, while Greer [1999] uses both record-level and field-level encodings. Unfortunately, field-level and record-level compression do not translate directly to the XML context. Goldstein et al. [1998] 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. [1998] applies compression to index structures such as B-trees and R-trees, to reduce their space usage, Westmann et al. [2000] discusses how to modify the relational query processor, the storage manager, and the query optimizer in the presence of field-level compression. Chen et al. [2000] focuses on query optimization for compressed relational databases by introducing *transient* decompression, that is, intermediary results are decompressed (e.g., in order to execute a join in the compressed domain), then recompressed for the rest of the execution. As XQueC does for XML data, both Chen et al. [2000] and Westmann et al. [2000] 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 Jagadish et al. [2004]. Finally, compression in a data warehouse setting has been applied in commercial DBMS products such as Oracle [Poess and Potapov 2003]. 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 Nonqueryable Compressors for XML Databases

XMill [Liefke and Suciu 2000] 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 recompressed with either GZIP or BZIP2 [2002].

XMLZIP [1999] 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 [Cheney 2001]. 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 Cheney [2005].

The previous three compressors 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 [Tolani and Haritsa 2002] 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, that 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 [Schmidt et al. 2002] benchmark queries).

XPRESS [Min et al. 2003] encodes whole paths into floating point numbers and, like XGrind, compresses textual (numeric, respectively) 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 [Min et al. 2006] 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 Buneman et al. [2003], 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* [Miklau and Suciu 2002], a rich subset of XPath. A more recent paper authored by Busatto et al. [2005] 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 [Cheng and Ng 2004] 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 [Ng et al. 2006] uses DTDs to perform compression and subsequent querying 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. Similar to

Table I. Comparative Analysis of Queryable XML Compressors

System	Struct./Text Compression	Homomorph.	Predicates	Language	Evaluation Strategies	Compression Granules
XGrind	Binary/Huffman + Dictionary	Yes	=, prefix	XPath subset	Top-down	Value/tag
XPRESS	RAE/Huffman (Arithmetic) + Dictionary + Difference	Yes	=,<, prefix	XPath subset ++	Top-down	Value/path
Buneman et al.	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

XQZip, the block size has to be carefully determined in order to achieve good performance.

Table I reports the major differences among the systems discussed. 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, that is, a container item or a tag which can be 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.

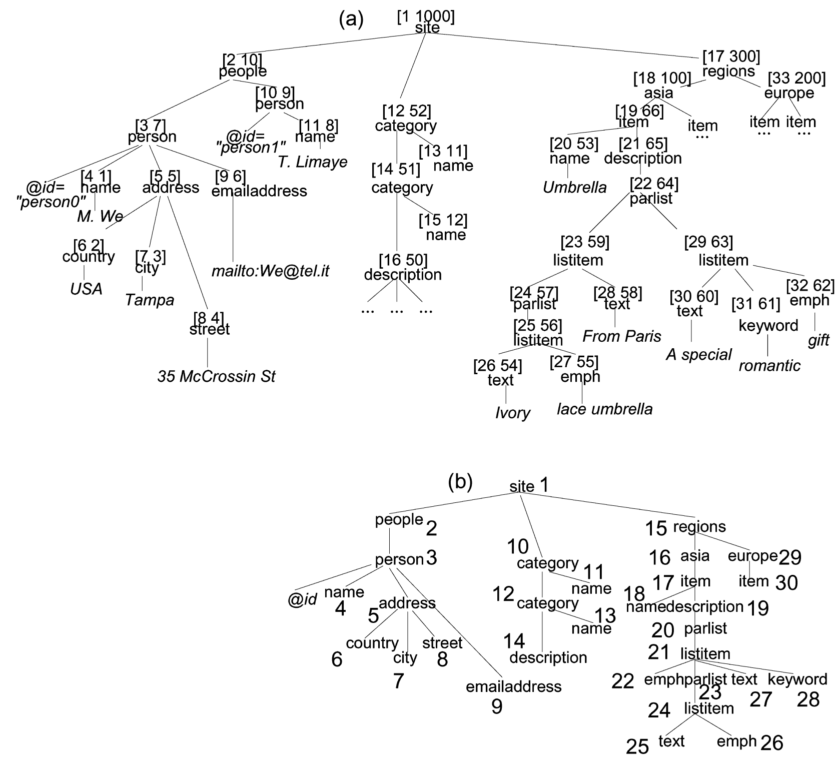
3. STORING AND QUERYING COMPRESSED XML DATA

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

3.1 XQueC Storage Structures and Architecture

XQueC splits an XML document into three data structures, depicted in Figure 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.

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,



(c)

/site	[1 1000]
/site/people	[2 11]
/site/people/person	[3 7][10 10]
/site/regions/asia/item	[19 66]

(d)

/site/people/person/@id	0 *c(person0)* 1 *c(person1)*
/site/people/person/name/#text	0 *c(M. Wile)* 1 *c(T. Limaye)*
/site/regions/asia/item/description/parlist/listitem/text/#text	0 *c(From Paris)* 1 *c(A special)*

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

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 Al-Khalifa et al. [2002], Halverson et al. [2003], Paparizos et al. [2003], and Grust [2002]. 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 Figure 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

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

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 Figure 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

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

Finally, the storage model includes a structure summary, that is, 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. Figure 1(b) depicts the structure summary for the sample document. The space usage of a summary SS is:

$$cs_{aux} = \sum_{n \in SS} \left(|tag(n)| + log_2(|SS|) \right), \tag{3}$$

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 Section 5), 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.

Figure 2 outlines 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 (see Section 4). 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 a suffix a call to the function *text()*, they return the text

[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.

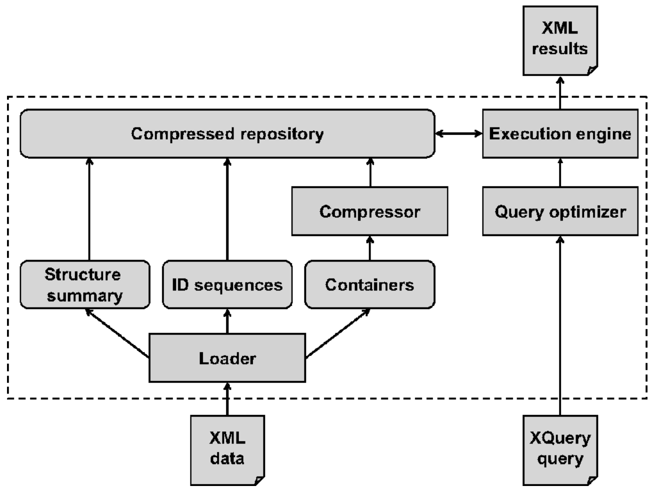


Fig. 2. Architecture of the XQueC prototype.

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 [XQUE 2004] 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) as $\mathcal{P}$, the 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 (respectively, descendant) pattern edges are shown by simple (respectively, 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

```
for $i in document("XMark.xml")//asia//item,
    $d in $i/description
where $d//keyword/text()="romantic"
return <gift><name>{$i/name/text()}<name>
            {$i//emph}
       </gift>
```

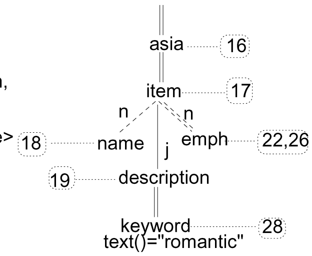


Fig. 3. Sample XQuery expression and its corresponding path-annotated query pattern.

should be output together for a given $i and $d match. The full pattern extraction algorithm, which is beyond the scope of this article, is described in Arion et al. [2006].

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 of 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 [Goldman and Widom 1997].

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 [Al-Khalifa et al. 2002] reflecting the semantics of pattern edges. We use structural outerjoins for optional edges as proposed in Chen et al. [2003]. 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 studied in Arion et al. [2006].

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 in the following.

Path partitioning provides for selective data access, more so than the tag-partitioning structural ID indexing used in Jagadish et al. [2002], Fiebig et al. [2002], and Halverson et al. [2003]. Node-partitioning schemes more aggressive than path partitioning can be envisioned [Buneman et al. 2003], but they may lead to excessive fragmentation. Structure Index Trees (SIT) [Cheng and Ng 2004] 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 [Al-Khalifa et al. 2002]. 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, and
—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 postorder 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 Jagadish et al. [2002], Fiebig et al. [2002], and Halverson et al. [2003], 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 Milo and Suciu [1999], and Goldman and Widom [1997], 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. The next section describes it in detail.

4. CHOICE OF COMPRESSION CONFIGURATIONS

Thus far we have discussed the utility of splitting the data instances into separate storage structures, that is, the containers and the tree structure. Container compression may become more efficient if appropriate container groups are considered and compressed together. There may exist multiple grouping choices which have a nontrivial impact on the size of compressed data and on the achievable query performance. As explained in the rest of this section, XQueC leverages a suitable cost model to drive the final choice.

4.1 Rationale for a Cost Model

The containers include a large share of the compressible XML data, that is, the values, thus making proper choices about compressing them is a key issue for an efficient XML compressor [Liefke and Suciu 2000].

Similar to other nonqueryable XML compressors, XQueC looks at the data commonalities to choose the container's compression algorithm. But how do we know that a compression algorithm is suitable for a container or a set of containers? In principle, we could use any eligible compression algorithm, but one with nice properties is, of course, preferable. Each algorithm has specific computational properties which may lead to different performance depending on the data sets actually used and on their similarities. In particular, the properties of interest for our purpose were the *decompression time*, which strongly influences the query response times over compressed data, the *compression factor* itself, and the *space usage of the source models* built by the algorithm. In fact, a container can be compressed individually or along with other containers; in the latter case, a group of containers share the same source model (i.e., the support structures used by the algorithm for both compressing and decompressing data). Grouping containers might be convenient, for example, when they exhibit high data similarity. Therefore, the space usage of the source model matters as much as the space usage of containers themselves and the decompression time; combining these three factors makes the choice even more challenging.

Besides the properties discussed, each compression algorithm is also characterized by the supported selections and/or joins in the compressed domain. There are several operations one can perform with strings, ranging from equality/inequality comparisons to prefix-matching and regular expression-matching. We give a brief classification of compression algorithms from the point of view of querying XML data. We distinguish among the following kinds of compressors.

—*Equality-preserving compressors*. These algorithms guarantee that equality selections and joins can be applied in the compressed domain. For instance, the Huffman algorithm supports both equality selections and equality joins in the compressed domain. The same holds for ALM, Extended Huffman [Moura et al. 2000], Arithmetic [Witten 1987], and Hu-Tucker [Hu and Tucker 1971].

—*Order-preserving compressors.* These algorithms guarantee that selections and joins using an inequality operator can be evaluated in the compressed domain. Examples of these algorithms are ALM, Hu-Tucker, and Arithmetic.

—*Prefix-preserving compressors*. These algorithms guarantee that prefix selections (such as c like pattern*) and joins (c_1 like c_2*) can be evaluated in the compressed domain. This property holds for the Huffman algorithm but does not hold for ALM.

—*Regular expression-preserving compressors.* These algorithms allow the evaluation of a selection of the form "c like *regular-expression*" in the compressed domain. Note that if an algorithm allows matching a regular expression, it also allows the determination of inequality selections as these can be equivalently expressed as regular expression selections. An example of an algorithm supporting regular expression selections is Extended Huffman.

The final choice of the algorithms to employ for the containers is driven by the predicates that are actually evaluated in the queries. The specific advantage of XQueC over similar XML compressors is that XQueC exploits query workloads to decide how to compress the containers in a way that supports efficient querying. Besides selection and join predicates, the cost model also takes into account top-level projections (i.e., those present in `RETURN` XQuery clauses) as they enforce the decompression of the corresponding containers. Query workloads have been already successfully employed in several performance studies from multiquery optimization to XML-to-relational mappings [Roy et al. 2000; Bohannon et al. 2002]. To the best of our knowledge, this is the first time they have been employed for deciding how to compress data.

We have so far discussed the multiple factors that influence the compression and querying performances. In the following, we illustrate this by means of an example.

4.1.1 *A Simple Case Study.* Let us consider three containers, namely c_1, c_2, and c_3, whose size are 500KB, 1MB, and 100MB, respectively. Assume that the workload features an inequality join between c_1 and c_2 and a prefix join between c_1 and c_3, whereas containers c_2 and c_3 are never compared by the workload queries (Figure 4(a)). To keep the example simple, we disregard top-level projections.

If we aim at minimizing only the storage costs (thus disregarding the decompression costs) among the multiple alternatives (i.e., keeping the containers separated versus aggregating them in all possible ways), we would prefer to compress each container separately (Figure 4(b)). Indeed, making groups of containers often increases both the sizes of compressed containers and source models because of the decreased intercontainers similarity within each group. In fact, if for instance c_1 and c_2 contain strings over two disjoint alphabets of two symbols each, and two separate source models are built, c_1 and c_2 are likely to be encoded with one bit per-symbol. If instead a single source model is used, two bits per-symbol are required, thus degrading the compression factor. A second relevant decision to be made is that of choosing the right algorithm for each

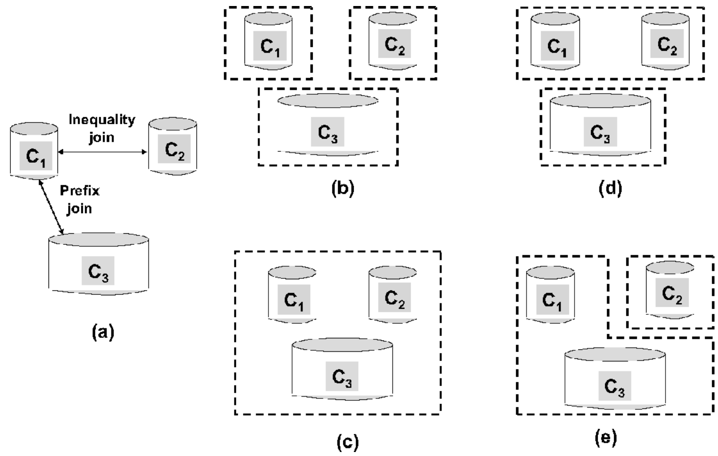


Fig. 4. Sample workload and possible partitioning alternatives.

separate container. Since only the storage cost matters, this algorithm should be the one with the best compression factor.

In contrast, if we aim at minimizing only the decompression costs but keeping the advantage of the reduced amount of data to be processed, then we would have to find a compression algorithm that supports both inequality and prefix joins in the compressed domain. If such an algorithm is available, the best choice is the one that aggregates all containers into one group, compressed with that algorithm (Figure 4(c)). Such a choice is optimal as it would nullify the decompression cost. Note that this is already in conflict with the choice of minimizing only the storage costs. If instead such an algorithm is not available, and there is one order-preserving algorithm for inequality joins and a prefix-preserving one for prefix joins, two possible alternatives arise (1) grouping c_1 together with c_2 and compressing them with the order-preserving algorithm, leaving c_3 as a singleton (2) or, grouping c_1 together with c_3 and compressing them with the prefix-preserving one, leaving c_2 as a singleton. The first choice saves decompression of a very large container, that is, c_3, thus making it preferable (Figure 4(d)).

The most general case is that of minimizing both storage and decompression costs. For the previous containers, there are again many possible alternatives. If the prefix-preserving algorithm matches the one that minimizes the storage costs, the choice of grouping is straightforward, leaving c_2 as a singleton (Figure 4(e)). On the other hand, if the two algorithms do not match or if the largest container is c_2, the scenario becomes increasingly more complex.

4.2 Costs of Compression Configurations

Our proposed cost model allows us to evaluate the cost of a given *compression configuration*, that is, a partition of the set of containers together with the assignment of a compression algorithm to each set in the partition. To do this, the cost model must also know the set of available compression algorithms (properly characterized with respect to certain types of comparison doable in the compressed domain) and the query workload.

More formally, we first define a *similarity matrix* F, that is, a symmetric matrix whose generic element $F_{i,j}$, with $0 \leq F_{i,j} \leq 1$, is the normalized similarity degree between containers c_i and c_j. A compression algorithm a is characterized by a tuple $\langle a.c_d(F), a.c_s(F), a.c_x(F,\sigma), a.\mathcal{L}\rangle$ where:

—the *decompression cost* $a.c_d(F)$ is a function estimating the cost of retrieving an uncompressed symbol from its compressed representation using algorithm a;
—the *storage cost* $a.c_s(F)$ is a function estimating the average cost of storing the compressed representation of a symbol using a;
—the *source model storage cost* $a.c_x(F,\sigma)$ is a function estimating the cost of storing the auxiliary structures needed to represent the source model of a set of containers sized σ using a;
—the algorithmic properties $a.\mathcal{L}$ are the kinds of comparisons supported by a in the compressed domain.

Note that each cost component is a function of the similarity among the containers. This is due to the fact that such costs always depend on the nature of data enclosed in the containers compressed together, that is, on the similarity among them (see the example in the previous section). Observe also that, as opposed to the containers storage cost, the source model storage cost is not symbol-specific, but it refers to an entire source model. This is due to the fact that the overhead of storing the source model is seldom linear with respect to the container's size [Moura et al. 2000].

The query workload $\mathcal{W}$, containing XQuery queries, is modeled using two sets, $cmp_{\mathcal{W}}$ and $proj_{\mathcal{W}}$, that reflect selections and joins among containers and top-level projections in $\mathcal{W}$:

—$cmp_{\mathcal{W}}$ is a set of tuples of the form $\langle q, i, j, l\rangle$, where $q \in \mathcal{W}$, $i \in \{1, \ldots, |\mathcal{C}|\}$, $j \in \{0, \ldots, |\mathcal{C}|\}$ are container indexes (index 0 represents constant values for selections), and $l \in \mathcal{L}$; each tuple denotes a comparison of kind l in q between containers c_i and c_j;
—$proj_{\mathcal{W}}$ is a set of tuples of the form $\langle q, i\rangle$, where $q \in \mathcal{W}$, and $i \in \{1, \ldots, |\mathcal{C}|\}$ is a container index; each tuple in $proj_{\mathcal{W}}$ denotes a projection on container c_i in q.

Note that $\mathcal{W}$ could easily be extended to provide information about the relative query frequency. For instance, suppose that a query q_1 features a join between containers c_1 and c_2, and a query q_2 has another join between containers c_3 and c_4. In such a case, the corresponding elements of $cmp_{\mathcal{W}}$ would be $\langle q_1, 1, 2, eq_j\rangle$ and $\langle q_2, 3, 4, eq_j\rangle$. If we also know from $\mathcal{W}$ that q_1 is three times more frequent than q_2, we simply add duplicates of $\langle q_1, 1, 2, eq_j\rangle$ in $cmp_{\mathcal{W}}$. This corresponds to viewing $cmp_{\mathcal{W}}$ as a bag instead of a set. The same applies to $proj_{\mathcal{W}}$.

Summarizing, the cost model input consists of (see Table II for the symbols used):

Table II. Summary of Symbols Used in the Cost Model

$\mathcal{C}$	*Set of textual containers*
$\mathcal{A}$	*Set of compression algorithms*
$\mathcal{W}$	*Query workload*
P	*Partition of* $\mathcal{C}$
p	*Set in* P
$\mathcal{L}$	*Kinds of comparisons considered*
alg	*Compression algorithm assignment function,* $P \to \mathcal{A}$
s	*Compression configuration* $\langle P, alg \rangle$
l	*Kind of comparison in* $\mathcal{L}$
a	*Algorithm in* $\mathcal{A}$
F	*Similarity matrix*
F_p	*Similarity matrix projected over the containers in* p
$a.c_d(F)$	*Cost of decompressing a symbol using the compression algorithm* a
$a.c_s(F)$	*Cost of storing a symbol using the compression algorithm* a
$a.c_x(F, \sigma)$	*Cost of storing the auxiliary structures for* σ *symbols using the compression algorithm* a
$cmp_{\mathcal{W}}$	*Set of comparisons in* $\mathcal{W}$
$proj_{\mathcal{W}}$	*Set of top-level projections in* $\mathcal{W}$
$d_{comp}(s, i, j, l)$	*Decompression cost due to a comparison of kind* l *between containers* c_i *and* c_j
$d_{proj}(q, s, i)$	*Decompression cost due to a projection in query* q *on container* c_i

—a set $\mathcal{C}$ of textual containers;
—a set $\mathcal{A}$ of compression algorithms;
—a query workload $\mathcal{W}$;
—a set $\mathcal{L}$ of *algorithmic properties*, denoting the kinds of comparisons considered;
—a compression configuration $s = \langle P, alg \rangle$, consisting of a partition P of $\mathcal{C}$, and a function $alg : P \to \mathcal{A}$ that associates a compression algorithm to each set in P.

The cost function, when evaluated on a configuration s, sums up different costs: the cost of decompression needed to evaluate comparisons and projections in $\mathcal{W}$, the compression factors of the different algorithms, and the cost of storing their source models. The overall cost of a configuration s with respect to a workload $\mathcal{W}$ is calculated as a weighted sum of the costs previously presented (sets $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{L}$ are implicit function parameters):

$$cost(s, \mathcal{W}) = \alpha \cdot decomp_{\mathcal{W}}(s) + \beta \cdot scc(s) + \gamma \cdot scm(s),$$

where $decomp_{\mathcal{W}}(s)$ represents the decompression cost incurred by s, $scc(s)$ represents the cost of storing the compressed data, $scm(s)$ represents the cost of storing the source models, and α, β, and γ, with $\alpha + \beta + \gamma = 1$, are suitable cost weights that measure the relative importance of the various components. Some manual intervention may occur here, that is to determine the actual values of these weights, which may depend on the application needs or the user preferences. In the following, we separately characterize each component of the cost function.

The containers storage cost for each set $p \in P$ is computed by multiplying the number of symbols in p by the storage cost incurred by the algorithm with which p is compressed. Such costs are influenced by the similarity among the containers in p so they are evaluated on the projection of $F_{\mathcal{C}}$ with respect to the containers in p (denoted as F_p). Thus, the containers storage cost is

$$scc(s) = \sum_{p \in P} \left(alg(p).c_s(F_p) \cdot \sum_{c \in p} |c| \right),$$

where $|c|$ denotes the total number of symbols appearing in container c. Similarly, the source model structure storage cost is

$$scm(s) = \sum_{p \in P} alg(p).c_x \left(F_p, \sum_{c \in p} |c| \right).$$

The decompression cost is evaluated by summing up the costs associated with both comparisons and projections in $\mathcal{W}$. To give an intuition, let us first consider a generic comparison occurring between two containers c_i and c_j. The associated decompression cost is zero if c_i and c_j share the same source model and the algorithm they are compressed with supports the required kind of comparisons in the compressed domain. A nonzero decompression cost occurs instead when one of the following conditions holds:

—c_i and c_j are compressed using different algorithms;
—c_i and c_j are compressed using the same algorithm but different source models;
—c_i and c_j are compressed using the same algorithm and the same source model, but the algorithm does not support the required kind of comparisons in the compressed domain.

For a selection over a container c_i, a zero decompression cost occurs only if the compression algorithm for c_i supports the required kind of selection in the compressed domain. In such a case, the constant value will be compressed using c_i's source model and the selection will be directly evaluated in the compressed domain. If instead the compression algorithm for c_i does not support the selection in the compressed domain, a nonzero decompression cost must be taken into account. To formalize this, we define a function d_{comp} that, given a compression configuration, calculates the cost of decompressing pairs of containers or single containers when involved in selections. The pseudocode for function d_{comp} is shown in Figure 5, where function $set(P, c)$ returns the set in P containing c. Similarly, function d_{proj}, given a compression configuration, calculates the decompression cost associated with the top-level projection of a container (Figure 5).

The overall decompression cost of a configuration s is computed by simply summing up the costs associated to each comparison and projection in the workload $\mathcal{W}$. The cost is therefore given by the following formula:

$$decomp_{\mathcal{W}}(s) = \sum_{\langle q,i,j,l \rangle \in cmp_{\mathcal{W}}} d_{comp}(s, i, j, l) + \sum_{\langle q,i \rangle \in proj_{\mathcal{W}}} d_{proj}(s, i).$$

function d_{comp}(s: compression configuration,
  $i \in \{1, \ldots, |\mathcal{C}|\}$ and $j \in \{0, \ldots, |\mathcal{C}|\}$: container indexes,
  $l \in \mathcal{L}$: comparison type): **return** a decompression cost
 If $j \neq 0$ // join predicate
  $p' \leftarrow set(P, c_i)$; $p'' \leftarrow set(P, c_j)$
  If $p' \neq p''$ Or $l \notin alg(p').\mathcal{L}$
   Return $|c_i| * alg(p').c_d(F_{p'}) + |c_j| * alg(p'').c_d(F_{p''})$
 Else // selection predicate
  $p \leftarrow set(P, c_i)$
  If $l \notin alg(p').\mathcal{L}$
   Return $|c_i| * alg(p).c_d(F_p)$
 Return 0

function d_{proj}(s: compression configuration,
  $i \in \{1, \ldots, |\mathcal{C}|\}$: container index): **return** a decompression cost
 $p \leftarrow set(P, c_i)$
 Return $|c_i| * alg(p).c_d(F_p)$

Fig. 5. Decompression cost for comparison predicates and top-level projections.

Note that, during the evaluation of $decomp_{\mathcal{W}}$, we keep track of the containers that have already been decompressed to make sure that the decompression cost of a container is taken into account only once.

4.3 Optimizing Compression Choices

The problem we deal with is that of finding the configuration incurring the minimum cost, provided the query workload ($\mathcal{W}$), a set of containers ($\mathcal{C}$), and a set of compression algorithms ($\mathcal{A}$). To the best of our knowledge, this problem (which in principle faces a search space of $\sum_{P \in \mathcal{P}} |\mathcal{A}|^{|P|}$, with $|\mathcal{P}|$ as the set of possible partitions of $\mathcal{C}$) cannot be reduced to any well-understood combinatorial optimization problem. Thus, we have designed some simple and fast heuristics that explore the search space to quickly find suitable compression configurations: a *Greedy* heuristic which starts from a naive initial configuration and makes local greedy optimizations; a *Group-based greedy* heuristic that adds a preliminary step to the previous one, aiming at improving the initial configuration; a *Clustering-based* heuristic that applies a classical clustering algorithm together with a cost-based distance measure. These heuristics are combined to obtain suitable compression configurations. This is feasible because all the heuristics are quite efficient in practice as we will show in Section 5.

4.3.1 *Greedy Heuristic.* We have devised a greedy heuristic that starts from a naive initial configuration, s_0 and improves over it by merging sets of containers in the partition. The main idea here is that of exploiting each comparison in $\mathcal{W}$ to enhance the current configuration; at each iteration, the heuristic picks the comparison that involves the maximum number of containers (improving over the heuristic presented in Arion et al. [2004] that randomized the choice of the comparison). Figure 6 shows the pseudocode of this heuristic. Steps 1–19 build the initial configuration by examining all the comparisons in the workload. Then, Steps 20–32 examine the cost of possible new configurations that are built by merging the groups obtained in previous steps but using a different

	function Greedy($\mathcal{W}$: query workload): **return** a compression configuration
1	$\mathcal{W}' \leftarrow \mathcal{W}$; $s_0 = \langle P_0, alg_0 \rangle$
2	Repeat
3	$c_i, c_j \leftarrow$ containers having the maximum number of comparisons in $\mathcal{W}'$
4	$\mathcal{W}' \leftarrow \mathcal{W}' \setminus \{$comparisons involving both c_i and $c_j\}$
5	If $\nexists p \in P_0 \mid c_i \in p$ or $c_j \in p$
6	add the set $p^n = \{c_i, c_j\}$ to P_0
7	$\mathcal{W} \leftarrow \mathcal{W} \setminus \{$comparisons involving both c_i and $c_j\}$
8	$A \leftarrow$ set of algorithms capable of doing the maximum number of comparisons
9	between c_i and c_j in the compressed domain
10	If $\lvert A \rvert = 1$
11	$a \leftarrow$ the algorithm in A
12	Else
13	$a \leftarrow$ the algorithm in A minimizing the expression
14	$\alpha \cdot a.c_d(F_{p^n}) + \beta \cdot a.c_s(F_{p^n}) + \gamma \cdot a.c_x(F_{p^n}, \sum_{c \in p^n} \lvert c \rvert)$
15	Make alg_0 associate p^n with a
16	until $\mathcal{W}' = \emptyset$
17	For each container $c \mid \nexists p \in P_0, c \in p$
18	$P_0 \leftarrow P_0 \cup \{c\}$
19	Make alg_0 associate $\{c\}$ with an algorithm a chosen as at line 8
20	$s_{curr} \leftarrow s_0$
21	Repeat
22	$pred \leftarrow$ predicate in $\mathcal{W}$ having the maximum number of occurrences
23	$c_i, c_j \leftarrow$ containers involved in $pred$
24	$p' \leftarrow set(P, c_i)$; $p'' \leftarrow set(P, c_j)$
25	$P' \leftarrow P_{curr} \setminus p' \setminus p'' \cup \{p' \cup p''\}$
26	For each $a_i \in \mathcal{A}$
27	$alg_{a_i} \leftarrow alg_{curr}$
28	Make alg_{a_i} associate p^u with a_i
29	$s_{a_i} \leftarrow \langle P', alg_{a_i} \rangle$
30	$s_{curr} \leftarrow argmin_{s \in \{s_{curr}, s_{a_1}, \ldots, s_{a_{\lvert \mathcal{A} \rvert}}\}} cost(s)$
31	$\mathcal{W} \leftarrow \mathcal{W} \setminus \{$ comparisons involving two containers in $p^u\}$
32	until $\mathcal{W} = \emptyset$
33	Return s_{curr}

Fig. 6. Greedy heuristic.

algorithm for them. The algorithm halts when all comparisons in the workload have been inspected.

4.3.2 *Group-Based Greedy Heuristic.* The group-based greedy heuristic is a variant of the greedy one and relies on the simple intuition that textual data marked by the same tag will likely have similar text content. Indeed, this heuristic treats groups of containers corresponding to paths ending with the same tag as a single container; this may lead to the building of a less trivial initial configuration than the one produced by the greedy heuristic. The latter is eventually applied on this initial configuration, thus, the pseudocode looks like the one in Figure 6 except for the preprocessing step.

4.3.3 *Clustering-Based Heuristic.* Since the problem of computing the compression configurations can be also thought of as a clustering problem, we

function Clustering($\mathcal{W}$: query workload): **return** a compression configuration
  $dist_{min}, dist_{max} \leftarrow$ minimum and maximum distances among two containers in $\mathcal{C}$
  Divide the range $[dist_{min}, dist_{max}]$ into equally-sized sub-ranges
  For each sub-range r
    If $\exists$ containers $c_i, c_j | dist(c_i, c_j) \in r$
      $P \leftarrow$ partition of $\mathcal{C}$ where containers c_i, c_j are in the same set
          only if $dist(c_i, c_j)$ is less or equal to the lowest value in r
      For each $p \in P$
        Make function alg associate p with algorithm a
            that minimizes $\alpha \cdot a.c_d(F_p) + \beta \cdot a.c_s(F_p) + \gamma \cdot a.c_x(F_p, \sum_{c \in p} |c|)$
      $s_{curr} \leftarrow argmin_{s \in \{s_{curr}, \langle P, alg \rangle\}} cost(s)$
  Return s_{curr}

Fig. 7. Clustering-based heuristic.

designed a heuristic that employs a simple clustering algorithm, that is, the *agglomerative single-link* algorithm [Jain et al. 1999]. In our case, the distance between pairs of containers must reflect the costs incurred when compressing those containers with different algorithms. This cost, in turn, depends on the containers' actual content. In particular, the distance between containers is proportional to the cost for decompressing the containers and storing them and their corresponding auxiliary structures. Moreover, for each algorithm, a non-null decompression cost occurs whenever the two compressed containers are involved in comparisons not supported by that compression algorithm (in the compressed domain). The distance can thus be formalized as follows:

$$dist(c_i, c_j) = \frac{\sum_{a \in \mathcal{A}} [\alpha \cdot u_{\mathcal{W}}(a, i, j) \cdot a.c_d(F_{\{c_i, c_j\}}) + \beta \cdot a.c_s(F_{\{c_i, c_j\}}) + \gamma \cdot a.c_x(F_{\{c_i, c_j\}}, |c_i| + |c_j|)]}{|\mathcal{A}|},$$

where $u_{\mathcal{W}}(a, i, j)$ is the number of comparisons in $\mathcal{W}$ between c_i and c_j that the algorithm a does not support in the compressed domain.

The pseudocode of the clustering-based heuristic is reported in Figure 7. At first, it chooses a number of distance levels among the containers. A distinct partition is generated for each distance level, letting the containers with distance less or equal to the chosen level be in the same set. This process leads creating partitions that have decreasing cardinality as the sets tend to be merged. Obviously, a singleton partition is eventually produced at a distance level greater than the maximum distance between containers. Since the cost function is invoked as many times as the number of distance levels, the chosen number of levels stems from a trade-off between execution times and probabilities of finding good configurations. Deciding the number of levels is empirically done and implies some manual tuning which is not required in the other heuristics. Finally, for each generated partition, the heuristic assigns to each set in the partition the algorithm that locally minimizes the costs.

5. EXPERIMENTAL ASSESSMENT

In this section, we present an experimental study of our storage and compression model.

Table III. XML Documents and Queries Used

Document d (MB)	# elems.	# tags	# conts.	# paths	ID bits	Provenance
DBLP (128)	3,332,129	40	136	125	44	[UWXML 2004]
INEX (483)	8,091,799	177	12,380	10,478	46	[INEX 2004]
NASA (24)	476,645	68	70	95	38	[UWXML 2004]
Shakespeare (7.5)	179,690	22	40	58	36	[IBIBLIO 2004]
SwissProt (109)	2,977,030	99	191	117	44	[UWXML 2004]
TreeBank (82)	2,437,665	250	220,818	338,748	44	[UWXML 2004]
UW course data (2.9)	84,051	18	12	18	34	[UWXML 2004]
XMark$n$ (n)	varies	varies	varies	varies	varies	[Schmidt et al. 2002]
XMark111 (111)	1,666,310	74	444	514	42	[Schmidt et al. 2002]
ShakespeareXPress (15.3)	359380	22	40	58	36	[IBIBLIO 2004]
1998statXPress (17.06)	422897	46	97	41	38	[IBIBLIO 2004]
WashingtonXPress (12.28)	336204	18	12	18	34	[UWXML 2004]

Query Code	Description
QXi	XMark query number i [Schmidt et al. 2002]
$QX1$	Point query.
$QX8$	Nested join query.
$QX14$	Regular-expression predicate query.
$QD1$	`FOR $p IN //person RETURN $p`
$QD2$	`FOR $h IN //homepage RETURN $h`
$QD3$	`FOR $a IN //address RETURN $a`

We present a set of performance measures, assessing the effectiveness of XQueC in different respects.

—Compression choices. We have evaluated the performance of the heuristics studied in Section 4 in partitioning the set of containers and choosing the right compression algorithm for each set in the partition.

—Compression factors. We have performed experiments on both synthetic and real-life data sets.

—Query execution times. We have probed XQueC query performance on XML benchmark queries [Schmidt et al. 2002] and the relative impact of decompression time on query performance.

We have implemented the XQueC system prototype in Java using Berkeley DB [BER 2003] as the backend that provides a set of low-level persistent storage structures. To store ID sequences and containers, we used Berkeley DB's persistent sequences: fixed length for ID sequences, and variable length for containers. At the physical level, we store the sequence of structural IDs in document order by using either a simple persistent sequence or persistent ordered-storage structure (e.g., a B+-tree).

Experimental setting. The name, size, and provenance of the used data sets are listed in Table III. The documents named XMark$n$ are generated using the XMark generator. For the purpose of comparison, we also include the same documents used in Min et al. [2003] (ShakespeareXPress, 1998statXPress, and WashingtonXPress). Table III also shows the queries used for experiments. All of our experiments have been performed on a machine with a 1.7GHz processor, 1GB RAM, and running Windows XP.

Table IV. Compression Configurations and Workloads Used

Configurations	Description
Cost-based	Blend of the heuristics presented in Section 4.2
NaiveX1 $X \in$ {Huffman,ALM}	One set with *all* string containers, apply compression algorithm X on the set
NaiveX2 $X \in$ {Huffman,ALM}	One set *for each* string container, apply compression algorithm X on each set
NaiveX3 $X \in$ {Huffman,ALM}	One set for each group of string containers whose paths end with the same tag; apply X on each set

Name	Comparisons	Projections	
XMark	155	63	Extracted from [Schmidt et al. 2002]
RW_1	217	42	Randomly generated over XMark75 (444 containers, out of which 426 contain strings).
RW_2	205	42	As above.

Other compression tools for comparison purposes. We discuss in the following the availability and usability of some competitors tools. XMill worked fine with all datasets but, as a nonqueryable compressor, it was only useful to compare compression factors. XGrind is aqueryable compressor, but we could only compare against its compression factors since the available query processor seems only capable of answering queries on documents sized as a few KB. Both XPRESS and XQZip are not publicly available and covered by copyright so we used the compression factors from Min et al. [2003] and Cheng and Ng [2004][2] in the comparison. However, in the XMark data sets used by Cheng and Ng [2004], the structure of rich textual types, such as item descriptions, has been eliminated. On XMark111, this leads to 7 instead of 12 levels of nesting. Finally, the queryable compressors described in Buneman et al. [2003] and Busatto et al. [2005] do not directly compare with XQueC since they do not produce a compressed persistent structure, and thus do not compress values.

For all the reasons described, a comprehensive comparison with other tools was not feasible. In contrast, we could make a comparison of our system with no compression with a compression-unaware XQuery engine, Galax 0.5.0 [Galax 2006] as shown in the remainder.

5.1 Compression Choices

In this section, we evaluate the heuristics presented in Section 4.2 by comparing the obtained compression configurations against the naive ones described in Table IV.

In our assessment of the cost model, we have adopted a possible characterization of the similarity matrix F[3]. To build the matrix, we have chosen the *Cosine* similarity function, defined as the cosine of the angle between the vectors that

[2]We also borrowed XGrind compression factors from Cheng and Ng [2004] as the latter's downloadable version was not usable for large datasets.
[3]Other characterizations of F and the corresponding cost functions are obviously possible but are beyond the scope of this article.

represent the containers. More formally, we first define the *signature* of a container as the number of occurrences of a fixed set of symbols Σ (composed of characters of the western alphabet plus some punctuation). Thus, the signature of a container c can be defined as a function $\overline{c} : \Sigma \rightarrow \mathbb{N}$. The cosine similarity between two containers c_i and c_j is therefore defined as follows:

$$F_{i,j} = \frac{\sum_{x \in \Sigma} \overline{c_i}(x) \cdot \overline{c_j}(x)}{\sqrt{\sum_{x \in \Sigma} \overline{c_i}(x)} \cdot \sqrt{\sum_{x \in \Sigma} \overline{c_j}(x)}}.$$

We have implemented two compression algorithms—ALM and Huffman—as a proof of concept in XQueC and, as a consequence, have based our experimental study on these algorithms. Two algorithms suffice to demonstrate our proof of concept. Indeed, more algorithms would only complicate the discussion, while not conveying new ideas. Moreover, the chosen two algorithms turn out to be quite appropriate as they fully cover XQuery [XQUE 2004] predicates in the compressed domain. We recall that the Huffman algorithm compressing one character at a time is relatively fast. It supports equality comparisons in the compressed domain and its compression dictionary is typically small. As a representative of order-preserving algorithms, we preferred ALM to other algorithms such as the Arithmetic and Hu-Tucker ones. Indeed, dictionary-based encoding has demonstrated its effectiveness with respect to other non-dictionary approaches [Moffat and Zobel 1992], and ALM outperforms Hu-Tucker [Antoshenkov et al. 1996]. Moreover, we have empirically chosen the number of distance levels used in the Clustering-based heuristic to be equal to 20.

Finally, as highlighted in Section 4, each compression algorithm is characterized by three functions that evaluate the costs of decompression ($c_d(F)$), of compressed container space usage ($c_s(F)$) and of auxiliary structures space usage ($c_x(F, \sigma)$). The costs of the compression algorithms have been measured on synthetic container filled with strings of up to 20 characters each; the total container sizes ranged from 100KB to 11MB, and the containers were generated with different cosine similarity values. Based on these measured values, we have calibrated the cost functions for ALM and Huffman algorithms.

We used three sample workloads, shown in Table IV: *XMark* is a subset of the XMark benchmark workload, while RW_1 and RW_2 were randomly generated based on the containers extracted from the same document. All the containers were extracted from XMark$_{75}$. We also analyzed a no-workload case to show the quality of compression results in the absence of a workload. In the experiments, we considered two possible assignments for the cost function weights: $\alpha = 1, \beta = 0, \gamma = 0$, for the case when only the decompression costs are taken into account; $\alpha = 0, \beta = 0.5, \gamma = 0.5$, where both the container and source model storage costs are taken into account and equally weighted.

We report the obtained results in Figure 8. We can observe that, in the majority of cases, the cost of the configuration obtained by running the heuristics is lower than the costs of the naive configurations. The difference in costs can be appreciated for all assignments of weights and all cases with/without workload.

P1: IAZ
ACMJ274-03 ACM-TRANSACTION May 6, 2007 12:10

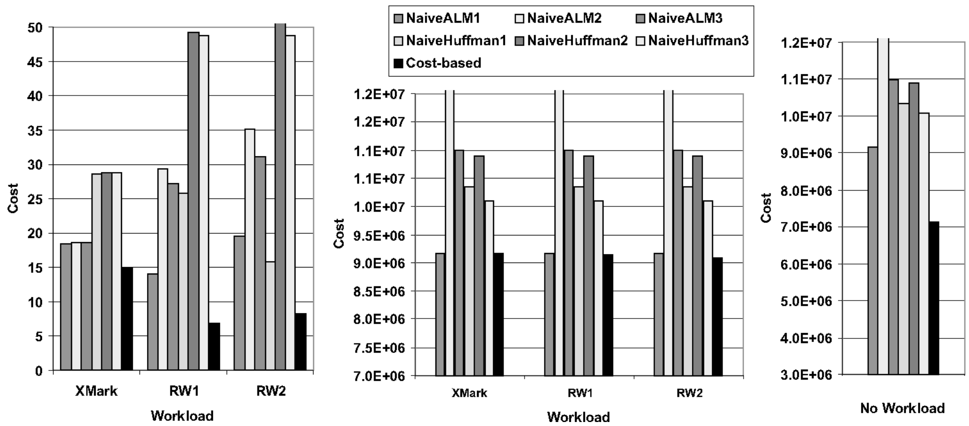


Fig. 8. Configuration costs with (a) $\alpha = 1, \beta = 0, \gamma = 0$; (b) $\alpha = 0, \beta = 0.5, \gamma = 0.5$; (c) $\alpha = 0, \beta = 0.5, \gamma = 0.5$, and no workload.

Moreover, as expected, the proposed heuristics turned out to be very fast; the maximum total execution time was 158.04 seconds for $\alpha = 1, \beta = 0, \gamma = 0$, 54.98 seconds for $\alpha = 0, \beta = 0.5, \gamma = 0.5$, and 8.08 seconds for the no-workload case.

5.2 Compression Performance

In this section we analyze XQueC compression performance by first showing the impact of our data structures on the compression factor and then measuring the latter with respect to competitors.

5.2.1 *Compression Factor Breakdown.* We start by showing how the various compressed data structures produced by XQueC impact the overall size of compressed documents. To ease readability, in Figure 9 we have used separate plots for the smallest documents (up to 25MB) and for the largest ones (up to 483MB). The relative importance of containers and structures varies with the nature of the considered document, for instance, TreeBank and Shakespeare do not have integer compressed containers. We can notice that, for most datasets, the compressed data structures reduce their size by a factor ranging between 2 and 4. Moreover, the size of the dictionary and the structure summary is also negligible in most cases. The results shown in Figure 9 are those obtained for *NaiveHuffman1*, one of the simplest configurations of Table IV.

Figure 10 shows the total size of compressed containers and dictionaries when varying the compression configurations. The configurations used here are built in the absence of a workload. The last column refers to a *none* compressor which isolates structure from content according to XQueC's model but stores the values as such (without any compression). The figure shows that the compression configuration impacts the resulting compressed structure sizes. In particular, among the naive configurations, those based on ALM tend to achieve the strongest container compression. The reason is that ALM exploits

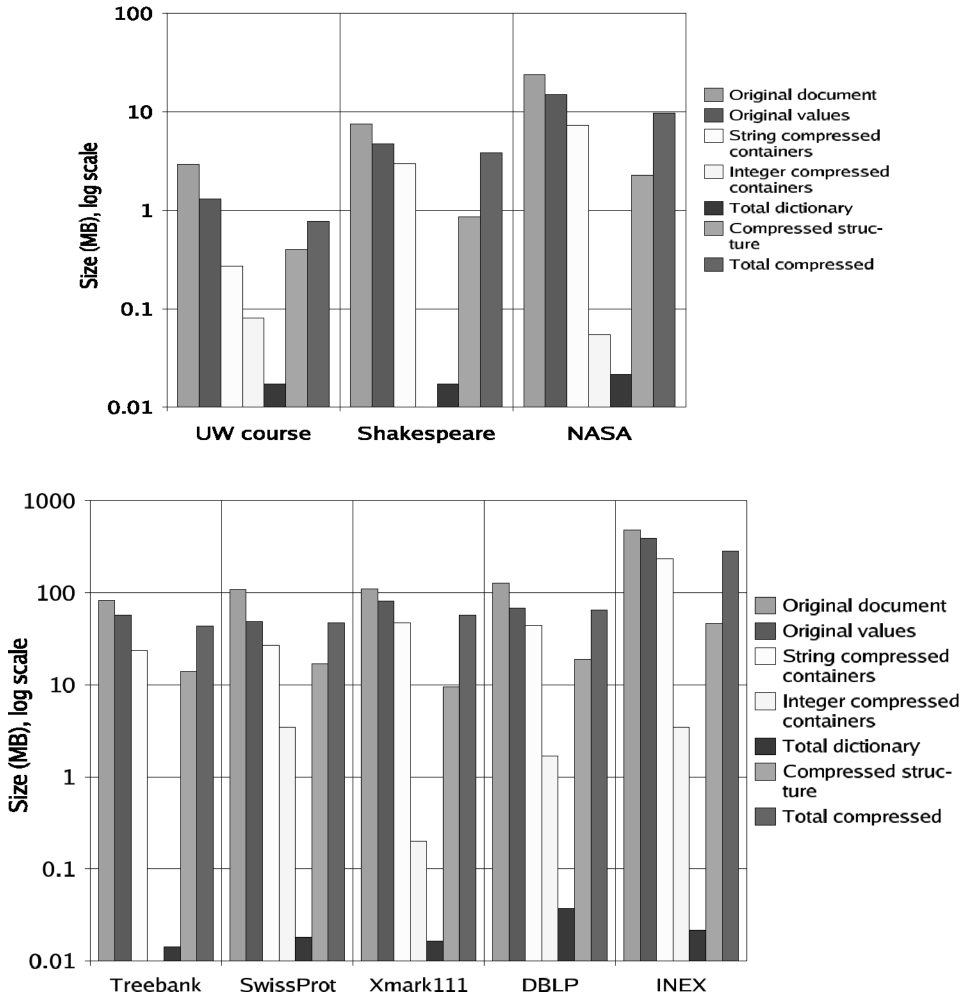
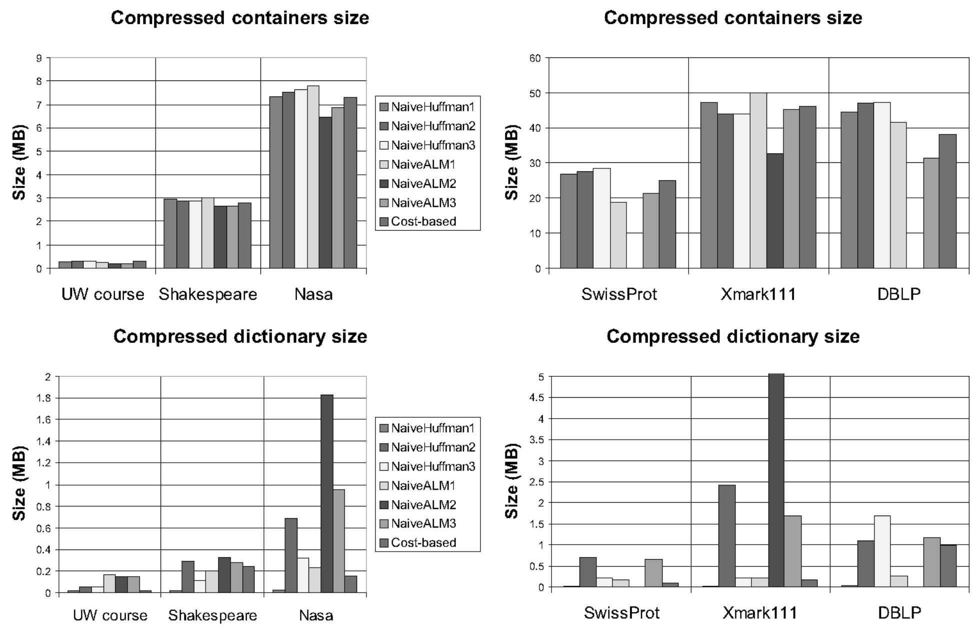


Fig. 9. Sizes of compressed data structures using configuration NaiveHuffman1.

repetitive substrings for compression. However, considering the dictionary size, *NaiveHuffman1* wins because it needs a single dictionary for all containers. Conversely, *NaiveHuffman3* and *NaiveHuffman2* are not as good as *NaiveHuffman1* since they require a separate dictionary for each container group. The same behavior occurs with ALM-based naive configurations. For instance, the dictionary size when using *NaiveALM2* reaches 1.8MB for the NASA document, against a compressed data size of 6.5MB. In such cases, the advantages of value compression may vanish. Moreover, for XMark111, the dictionary size reached a value of 11.7MB (for readability, the graph is capped at 5MB), whereas for SwissProt and DBLP, the compressor's memory requirements were so high that loading failed. The none compressor itself achieves a light compression due to the fact that opening and closing tags are simply replaced with (sequences of) pre and post ID values.



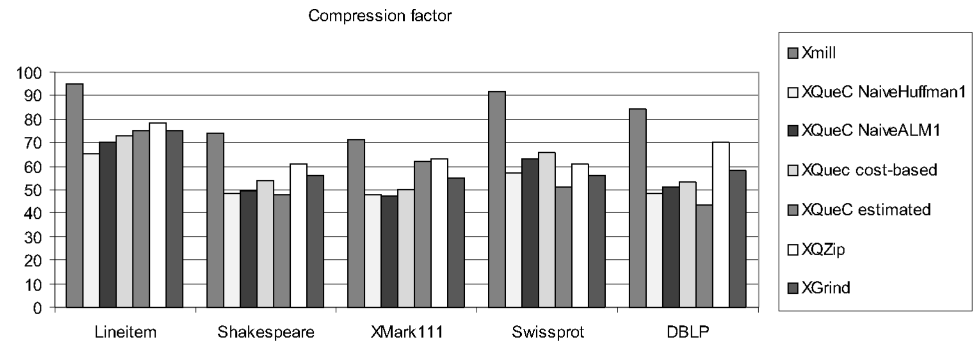
Comp. factor	NH1	NH2	NH3	NALM1	NALM2	NALM3	Cost-based	None
UW courses	**0.74**	0.71	0.71	0.7	0.72	0.72	**0.74**	0.42
Shakespeare	**0.49**	0.47	**0.49**	0.45	**0.49**	**0.49**	0.48	0.26
NASA	**0.59**	0.56	0.57	0.57	0.56	0.58	**0.59**	0.27
SwissProt	0.57	0.55	0.55	**0.64**	-	0.61	0.61	0.4
XMark111	0.48	0.49	0.51	0.45	**0.61**	0.49	0.49	0.18
DBLP	0.49	0.46	0.45	0.51	-	**0.58**	0.55	0.32

Fig. 10. Compressed string containers, dictionary sizes, and compression factors for various compression configurations (loading failed when using *NaiveALM2* on both SwissProt and DBLP datasets).

The results show that *NaiveHuffman1*, among all other naive configurations, reaches a fairly good compromise between compression ratios and times. Thus, in the remainder of the experimental study, *NaiveHuffman1* will often be adopted as a baseline configuration.

These experiments also show that a cost-based search, blending the conflicting needs of small containers (obtained by using small container groups and ALM), and of small dictionaries (by using large container groups and Huffman) is quite effective overall. We see that the cost-based compression factor is close to the best CF recorded (shown in bold in Figure 10) and quite robust, whereas naive strategies, attractive on some documents (e.g., *NaiveALM2*) are plainly unfeasible on other documents. Good trade-offs are harder to find when multiplying the available compression algorithms, thus the interest of a cost-based search method.

5.2.2 *Compression Factor Compared with Other XML Compressors.* We now measure the XQueC compression factor (CF) and compare it with that



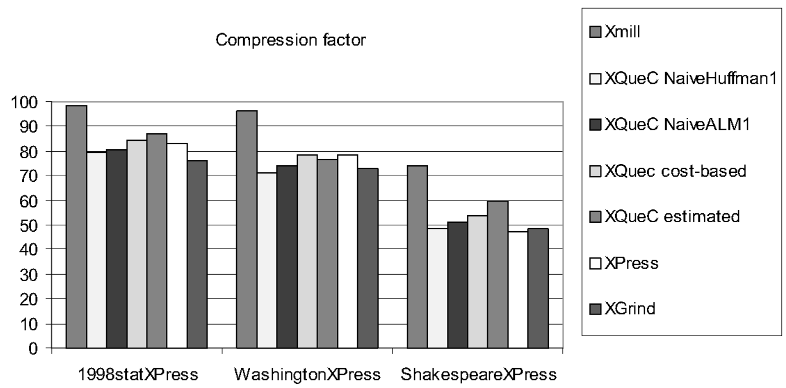
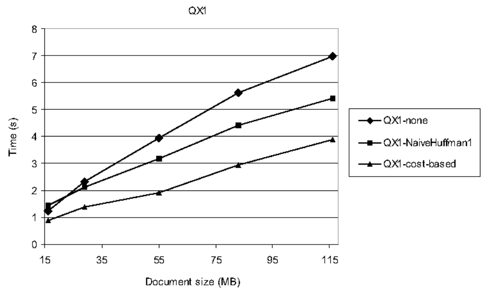
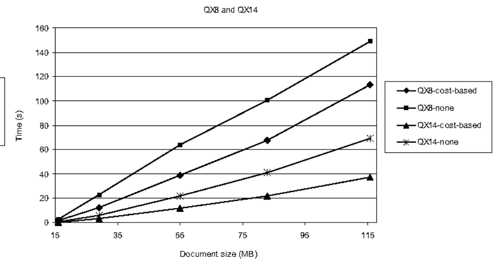


Fig. 11. XQueC CF compared with its competitors.

of competitor systems (within the limitations discussed). We have divided the experiments into two parts, depending on the compared competitors CFs. Figure 11 (top) shows the first comparison, that is, XQueC CF against those of XQZip (as reported in Cheng and Ng [2004]), those of XMill (which we computed ourselves) and those of XGrind (also as reported in Cheng and Ng [2004]). We report the obtained results for the *NaiveHuffman1*, *NaiveALM1*, and *cost-based* configurations; we also report the cost-based estimates computed by the cost model for the cost-based configurations. It can be noticed that the cost-based configurations always overcome the naive ones, and that the estimate obtained via the cost model is acceptably sharp. Although XQueC CF is slightly inferior to that of XQZip and XGrind, the small difference is balanced with XQueC's greater query capabilities.

Secondly, we show the XQueC CFs against those of XPRESS, XMill, and XGrind. We used the same datasets as in Min et al. [2003] and compared them with the compression factors reported in that paper. Figure 11 (bottom) shows that XQueC CFs are rather comparable to those of XPRESS and slightly worse than XGrind. We recall that these datasets have all been obtained, as in Min et al. [2003], by multiplying the original data sources several times; however, this operation does not give any advantage to our compression techniques whose inherent properties do not allow them to recognize the presence of an entire repeated document.





	XMark15		XMark30		XMark55		XMark83		XMark111	
Query	XQueC	Galax	XQueC	Galax	XQueC	Galax	XQueC	Galax	XQueC	Galax
$QX1$	1.25s	7.28s	2.33s	14.43s	3.95s	32.16s	5.62s	56.02s	6.97s	1m24.01s
$QX8$	2.8s	7.60s	22.9s	14.91s	1m3.8s	32.78s	1m40.5s	1m1.47s	2m29s	1m32.17s
$QX14$	0.3s	21.36s	5.8s	47.77s	22s	2m8.84s	41s	3m49.30s	1m9s	>10m

Fig. 12. Evaluation times for XMark queries (top); actual numbers for XQueC 'none' and for Galax (bottom).

5.3 Query Execution Times

In this section, we assess XQueC query evaluation performance.

5.3.1 *Query Performance.* We study the scaleup of the XQueC query engine with various document sizes and the impact of cost-based compression configurations on query execution times. Notice that we could not compare to other XML queryable compressors (as explained earlier), whereas we could report comparative execution times for an XQuery compression-unaware implementation[Galax 2006].

We start by showing experiments on XQueC query performance. In Figure 12 (left), we show the results obtained running XMark query $QX1$ on XMark documents, using three configurations: *NaiveHuffman1*, as a baseline, the cost-based one, and the one using no compression. We notice that the cost-based configuration leads to an average improvement of 55.2% with respect to *NaiveHuffman1*. In addition, query time scales linearly with the document size for query $QX1$. Measures with other XMark queries showed the same trend. We report in a separate figure (Figure 12, right) the results of $QX8$ and $QX14$ for the cost-based and none configurations, whereas the *NaiveHuffman1* is omitted to avoid clutter. $QX14$ is a selection query with a regular-expression predicate, whereas $QX8$ is a more complex nested join query. For such representative queries of the XMark benchmark, we also obtained a linear scaleup, thus confirming XQueC scalability.

For convenience, the table of Figure 12 reports the XQueC execution times under none configuration for queries $QX1$, $QX8$, and $QX14$, and the Galax [2006] times for the same queries. Although the two XQuery engines cannot be absolutely compared due to many differences in the implementations, we just want to note that the performance of our system stays competitive when compression is not employed. Comparable results, obtained with the queries QD_1, QD_2, QD_3 described next, are omitted for space reasons.

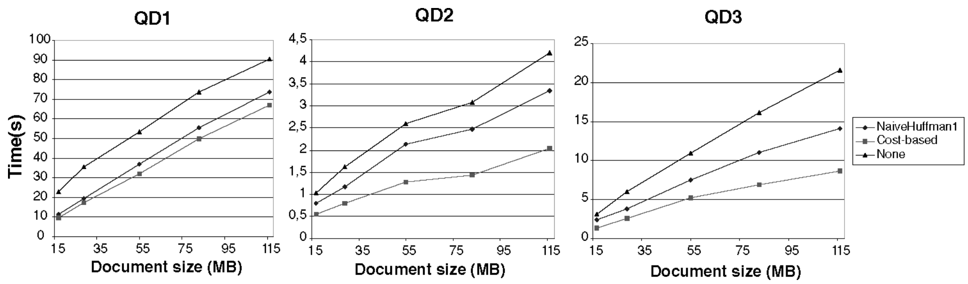


Fig. 13. Evaluation times for reconstruction queries.

5.3.2 *Decompression Time.* In this section, we examine the impact of data decompression on the effort required to construct complex query results. Indeed, reconstructing the query results for compressed data is more time-consuming than for the uncompressed case. A first experiment is aimed at examining the impact of the naive and cost-based compression configurations on the execution time of three ad-hoc selective XQuery queries with descendant axis. These queries, illustrated in Table III, are representative of various cases of reconstruction. In particular, *QD1* returns about 1/10th of the input document, while *QD*2 is more selective, and *QD*3 returns deep XML fragments with complex structure. Figure 13 shows the results obtained by running the queries against different XMark documents. We compare the configuration obtained by the cost-based search with the baseline *NaiveHuffman1* and none configurations. The plots in Figure 13 show that XQueC total decompression time grows linearly with the document size and emphasize the advantages of cost-based search over naive and none configurations.

Finally, Figure 14 (top) reports the time needed to read and decompress containers from two datasets having comparable size but different structure: XMark17 and Shakespeare. We consider two different configurations: *NaiveHuffman1* and *NaiveALM1*. The figure shows that, due to a slightly better compression ratio, the time to read data from disk is smaller for the *NaiveHuffman1* configuration. At the same time, character-based Huffman decompression is quite slow when compared with ALM symbol-based decompression. Therefore, the overall time is minimized by using ALM. This confirms the utility of properly modeling the costs of the possibly different compression configurations with two algorithms such as ALM and Huffman. Indeed, ALM turns out to be used by our heuristics in most of the cases; presumably, Huffman might be preferred if compression time was also taken into account. Secondly, decompression time is more important on the XMark document when compared to the Shakespeare one. This can be explained by the fact that Shakespeare tends to have relatively short strings (lines exhibiting bounded length) as opposed to the longer strings present in XMark. Figure 14 (bottom) shows that the same trend is obtained with larger documents: Nasa, SwissProt, DBLP, XMark55, XMark83, and XMark111.

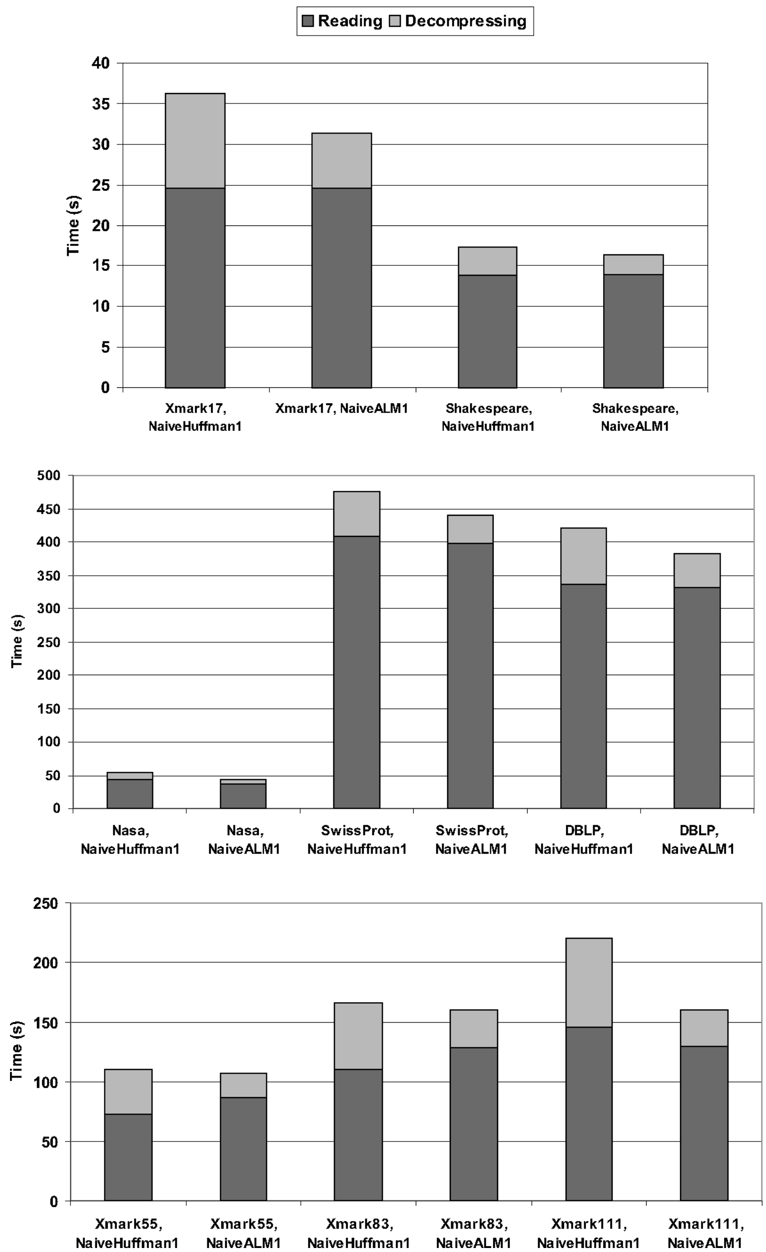


Fig. 14. Time for reading and decompressing containers.

5.4 Lessons Learned

Our experiments have studied several aspects of the XQueC system. First, we have assessed the utility of the proposed heuristics at finding suitable solutions when compared with the naive strategies. Not only is a cost-based solution less expensive, but it is also faster than the naive ones. Next we have examined the compression and querying capabilities of our system, establishing the utility of cost-based configurations. By means of selected naive configurations that we chose as baselines, we were able to pinpoint the advantages of using our cost model. In particular, the compression factor obtained with the cost-based configurations is, within the majority of the datasets, the best one recorded with a naive configuration, thus confirming that the cost-based search is effective. In contrast, picking a naive configuration at random and using it for compressing the datasets may sometimes be unfeasible. In the worst case, we would be forced to exhaustively compute the compression factors for an arbitrary number of naive configurations: such a number becomes higher as the number of compression algorithms increases. Third, we have demonstrated the scalability of the query engine using the XMark benchmark. We have measured the evaluation times of a significant set of XMark queries and showed the reconstruction times for increasingly selective XQuery queries. The results thus obtained demonstrate that the combination of proper compression strategies with a vertically fragmented storage model and efficient operators can prove successful. Moreover, the cost-based configurations performs better for queries than the naive ones, thus highlighting the importance of a cost-based search. By means of a no-compression version of XQueC, we were also able to compare with a compression-unaware XQuery implementation and show that we are competitive. Finally, we have verified that, during query processing, the time spent for reading and decompressing containers can vary depending on the algorithm and the datasets, thus leading to blend these factors in a suitable cost computation.

6. CONCLUSIONS

The XQueC approach is to seamlessly bring compression into XML databases. In light of this, XQueC is the first XML compression and querying system supporting complex XQuery queries over compressed data. XQueC uses a persistent store and produces an actual disk-resident image, thus it is able to handle very large datasets and expressive queries. Moreover, a cost-based search helps identify the compression partitions and their corresponding algorithms. We have shown that XQueC achieves a reasonable reduction of document storage costs because of its ability to efficiently process queries in the compressed domain.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for comments that helped strengthen the article. We are grateful to Michael Benedikt for giving us suggestions on the writing. We are indebted to our students Gianni Costa and Sandra D'Aguanno for their contribution to the former prototype described in Arion et al. [2004], and to Erika De Francesco for her contribution on a new implementation of the ALM algorithm.

REFERENCES

AL-KHALIFA, S., JAGADISH, H., PATEL, J., WU, Y., KOUDAS, N., AND SRIVASTAVA, D. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE, 141–152.

AMER-YAHIA, S. AND JOHNSON, T. 2000. Optimizing queries on compressed bitmaps. In *Proceedings of 26th International Conference on Very Large Data Bases*. ACM, 329–338.

ANTOSHENKOV, G. 1997. Dictionary-based order-preserving string compression. *VLDB J. 6*, 1, 26–39.

ANTOSHENKOV, G., LOMET, D., AND MURRAY, J. 1996. Order preserving string compression. In *Proceedings of the 12th International Conference on Data Engineering*. IEEE, 655–663.

APA 2004. Apache custom log format. http://www.apache.org/docs/mod/mod_log_config.html.

ARION, A., BENZAKEN, V., MANOLESCU, I., PAPAKONSTANTINOU, Y., AND VIJAY, R. 2006. Algebra-based identification of tree patterns in XQuery. In *Proceedings of the International Conference on Flexible Query Answering Systems*. 13–25.

ARION, A., BONIFATI, A., COSTA, G., D'AGUANNO, S., MANOLESCU, I., AND PUGLIESE, A. 2004. Efficient query evaluation over compressed XML data. In *Proceedings of the International Conference on Extending Database Technologies*. Heraklion, Grece, 200–218.

ARION, A., BONIFATI, A., MANOLESCU, I., AND PUGLIESE, A. 2006. Path summaries and path partitioning in modern XML databases. In *Proceedings of the International World Wide Web Conference*. 1077–1078.

BER 2003. Berkeley DB Data Store. http://www.sleepycat.com/products/data.shtml.

BOHANNON, P., FREIRE, J., ROY, P., AND SIMEON, J. 2002. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE, 64–76.

BUNEMAN, P., GROHE, M., AND KOCH, C. 2003. Path queries on compressed XML . In *Proceedings of 29th International Conference on Very Large Data Bases*. Morgan Kaufmann, 141–152.

BUSATTO, G., LOHREY, M., AND MANETH, S. 2005. Efficient memory representation of XML documents. In *Proceedings of the 10th International Symposium on Database Programming Languages.* Trondheim, Norway, 199–216.

BZIP2 2002. The bzip2 and libbzip2 Official Home Page. http://sources.redhat.com/bzip2/.

CHEN, Z., GEHRKE, J., AND KORN, F. 2000. Query optimization in compressed database systems. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, 271–282.

CHEN, Z., JAGADISH, H., LAKSHMANAN, L., AND PAPARIZOS, S. 2003. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *Proceedings of 29th International Conference on Very Large Data Bases*. Morgan Kaufmann, 237–248.

CHENEY, J. 2001. Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference*. IEEE Computer Society, 163–172.

CHENEY, J. 2005. An empirical evaluation of simple DTD-conscious compression techniques. In *WebDB*. 43–48.

CHENG, J. AND NG, W. 2004. XQzip: Querying compressed XML using structural indexing. In *Proceedings of the International Conference on Extending Database Technologies*. Heraklion, Greece, 219–236.

FIEBIG, T., HELMER, S., KANNE, C., MOERKOTTE, G., NEUMANN, J., SCHIELE, R., AND WESTMANN, T. 2002. Anatomy of a native XML base management system. *VLDB J. 11*, 4, 292–314.

GALAX 2006. Galax: An implementation of XQuery. Available at www.galaxquery.org.

GOLDMAN, R. AND WIDOM, J. 1997. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*. Morgan Kaufman, 436–445.

GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. 1998. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering*. IEEE, 370–379.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Compu. Surv. 25*, 2, 73–170.

GREER, R. 1999. Daytona and the fourth-generation language Cymbal. In *Proceedings ACM SIGMOD International Conference on Management of Data*. ACM, 525–526.

GRUST, T. 2002. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 109–120.

HALVERSON, A., BURGER, J., GALANIS, L., KINI, A., KRISHNAMURTHY, R., RAO, A., TIAN, F., VIGLAS, S., WANG, Y., NAUGHTON, J., AND DEWITT, D. 2003. Mixed mode XML query processing. In *Proceedings of 29th International Conference on Very Large Data Bases*. Morgan Kaufmann, 225–236.

HU, T. C. AND TUCKER, A. C. 1971. Optimal computer search trees and variable-length alphabetical codes. *SIAM J. of App. Mathe. 21*, 4, 514–532.

HUFFMAN, D. A. 1952. A method for construction of minimum-redundancy codes. In *Proceedings of the IRE*. 1098–1101.

IBIBLIO 2004. Ibiblio.org web site. Available at www.ibiblio.org/xml/books/biblegold/examples/baseball/.

INEX 2004. *IN*itiative for the *E*valuation of *X*ML retrieval. inex.is.informatik.uni-duisburg.de:2004.

JAGADISH, H. V., AL-KHALIFA, S., CHAPMAN, A., LAKSHMANAN, L. V., NIERMAN, A., PAPARIZOS, S., PATEL, J., SRIVASTAVA, D., WIWATWATTANA, N., WU, Y., AND YU., C. 2002. Timber: A native XML database. *VLDB J. 11*, 4, 274–291.

JAGADISH, H. V., NG, R., OOI, B. C., AND TUNG, A. K. H. 2004. ItCompress: An iterative semantic compression algorithm. In *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society, 646–658.

JAIN, A. K., MURTY, M. N., AND FLYNN, P. J. 1999. Data clustering: A review. *ACM Comput. Surv. 31*, 3, 264–323.

LIEFKE, H. AND SUCIU, D. 2000. XMILL: An efficient compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, 153–164.

MIKLAU, G. AND SUCIU, D. 2002. Containment and equivalence for an XPath fragment. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Conference on the Principles of Database Systems*. 65–76.

MILO, T. AND SUCIU, D. 1999. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory (ICDT)*. 277–295.

MIN, J. K., PARK, M., AND CHUNG, C. 2003. XPRESS: A queriable compression for XML data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 122–133.

MIN, J. K., PARK, M., AND CHUNG, C. 2006. A compressor for effective archiving, retrieval, and update of XML documents. *ACM Trans. Intern. Techn. 6,* 3.

MOFFAT, A. AND ZOBEL, J. 1992. Coding for compression in full-text retrieval systems. In *Proceedings of the Data Compression Conference (DCC)*. 72–81.

MOURA, E. D., NAVARRO, G., ZIVIANI, N., AND BAEZA-YATES, R. 2000. Fast and flexible word searching on compressed text. *ACM Trans. Inform. Syst. 18*, 2 (April), 113–139.

NG, W., LAM, Y. W., AND CHENG, J. 2006. Comparative analysis of XML compression technologies. *WWW J. 9*, 1, 5–33.

NG, W., LAM, Y. W., WOOD, P., AND LEVENE, M. 2006. XCQ: A queriable XML compression system. *Inter. J. Knowl. Inform. Syst.* To appear.

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., AND YU, C. 2003. TIMBER: A native system for querying XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 672.

POESS, M. AND POTAPOV, D. 2003. Data compression in Oracle. In *Proceedings of 29th International Conference on Very Large Data Bases*. Morgan Kaufmann, 937–947.

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data.* Dallas, TX. 249–260.

SCHMIDT, A., WAAS, F., KERSTEN, M., CAREY, M., MANOLESCU, I., AND BUSSE, R. 2002. XMark: A benchmark for XML data management. In *Proceedings of 28th International Conference on Very Large Data Bases*. Morgan Kaufmann, 974–985.

TOLANI, P. AND HARITSA, J. 2002. XGRIND: A Query-friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE, 225–235.

Transaction Processing Performance Council. 1999. TPC-H benchmark database. http://www.tcp.org.

UWXML 2004. University of Washington's XML repository. Available at www.cs.washington.edu/research/xmldatasets.

Westmann, T., Kossmann, D., Helmer, S., and Moerkotte, G. 2000. The implementation and performance of compressed databases. *ACM SIGMOD Rec. 29*, 3, 55–67.

Witten, I. H. 1987. Arithmetic coding For data compression. *Comm. ACM*, 857–865.

XMLZIP 1999. XMLZip XML compressor. Available at http://www.xmls.com/products/xmlzip/xmlzip.html.

XQUE 2004. The XML query language. http://www.w3.org/XML/Query.

Received December 2005; revised January 2006; accepted October 2006