XML compression is becoming attractive these last years, as it reduces the disadvantage of “verbose” data representation in XML. While in many applications, such as data exchange and data archiving, compressing and decompressing one document at a time is sufficient, in applications where queries must be run over the compressed documents, decompression is an extra effort for the query processor, and its performance penalty may overcome data compression benefits. Smoothly balancing the interests of compression and query processing received significant attention in the domain of relational databases. However, these results do not directly translate to XML data.

In this paper, we address the problem of seamlessly embedding compression in XML databases, without degrading query processing performance. This opens up several new interesting questions. The choice of compression granularity and compression algorithms has to be revisited, since the setting is not the same as for relational databases. Executing queries as much as possible in the compressed domain has to be rethought in the framework of XML query processing, due to the richer structure of XML data. Indeed, a proper storage design plays a crucial role here.

Our XQueC system (standing for XQuery Processor and Compressor) 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 assessment is also 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: XML databases, XML compression

Additional Key Words and Phrases: XML data management, XML compression, XQuery
1. INTRODUCTION

An increasing amount of data on the Web is now available as XML, either being directly created in this format, or exported to XML from other formats. XML is flexible, self-describing, and accommodates well structured textual data. However, 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 significantly its volume. For example, [Liefke and Suciu 2000] shows that specific format data, such as Weblog data [APA 2004] and SwissProt data [Swissprot], once XML-ized, grow by about 40% of their original size.

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 a document for each query evaluation, since this would penalize query performance. Instead, decompression must be carefully applied on the minimal amount of data needed for each query.

In this paper, we present the XQueC system, that allows compressing and querying XML data, decompressing it only if needed. This idea has been introduced as lazy decompression in relational databases [Chen et al. 2000; Westmann et al. 2000], that is, decompress as little (and late) as possible. However, lazy decompression is not easily incorporated into existing XML compressors, such as XMill [Liefke and Suciu 2000].

Existing queryable XML compressors, such as XGrind [Tolani and Haritsa 2002], XPRESS [Min et al. 2003] and XQZip [Cheng and Ng 2004] only partially address this problem, since their query processing capabilities in the compressed domains are quite restricted. This is mostly due to their compression model, which is not transparent enough to process complex queries, and also to the choices of individual data compression algorithms they use (more details on this comparison are provided in Section 2).

Thus, the choice of the data compression granularity, compressed storage format, and actual compression algorithms strongly impacts the balance between compression performance, query expressive power in the compressed domain, and query performance. We now outline XQueC’s approach in this area.

The compression granules Choosing the compression granules carries out a double-fold meaning in the realm of XML. The compression granules represent the units of compressions that the query processor deals with, but at the same time embody the storage units manipulated at the physical level.

Studies on compressing and querying relational data are not directly usable for XML, as the XML data is much more textual, and carries a richer structure not easily captured by relational formats [Krishnamurthy et al. 2003]. Furthermore, the compression granularity in relational databases may range from file-level to record-level, field-level and block-level (see [Westmann et al. 2000]), but these granules do not translate directly to XML storage, since XML disk-based storage does not provide such clear granularity units as “field”, “record” and “block”. For all these reasons, XQueC proposes a compression-compliant storage for XML data, preserving many possible options for the query processor. Previous XML query-aware compressors, such as XGrind [Tolani and Haritsa 2002], XPRESS [Min et al. 2003] and XQZip [Cheng and Ng 2004], were not equipped with a storage back-end, instead worked by entirely or partially constructing in memory the XML data tree.
XQueC’s data fragmentation strategy is borrowed from XMill [Liefke and Suciu 2000], based on separating XML document structure and contents. XMill made the important observation that data nodes (leaves of the XML tree) found on the same path in an XML document (e.g., /site/people/person/address/city in the XMark [Schmidt et al. 2002] documents) often exhibit similar content. Therefore, it makes sense to group all such values into a single container and choose the compression strategy once per container. Subsequently, XMill treated a container like a bulk “chunk of data” and compressed it as such, which disables access to any individual data node. Separately, XMill compressed and stored the structure tree of the XML document.

XMill cluster values found on one or several paths, into a container. The default is one container per path, but users can manually specify other choices. In contrast, XQueC builds a container for each distinct path, and automatically analyzes their similarities to make intelligent compression choices. Most importantly, unlike XMill, each container item is individually compressed and individually accessible. Another important component of our storage design is the structure summary, which is similar to a Dataguide [Goldman and Widom 1997]. While XMill uses a dataguide to fully reconstruct the document, we also use it for query evaluation in the compressed domain.

Comparison-enabled compression algorithms

An important issue is to choose the right compression algorithm(s) for the containers. Since the PCDATA content, once compressed, affects the final document compression ratio more than the structure itself, the containers compression is crucial. We found out that, after dictionary-encoding document tags, the structure typically takes no more than 20%-30% of the document size.

Regarding values compression, projects like XGrind [Tolani and Haritsa 2002] and XPRESS [Min et al. 2003] do not consider the problem of further aggregating the containers based on their commonalities (see also Section 2). XQueC detects such containers and compresses them with the same algorithm and the same source model 1. The source model may affect the choice of the compression algorithm as well as the containers occupancy and the decompression times. We are mostly concerned here with textual containers, since they seem to be much better represented in XML than in e.g., relational databases.

We can summarize the previous problem as follows: given a set of textual containers, how to group them and how to compress each container group?

This problem is addressed in XQueC by looking at the query workload. Indeed, what really matters in the above choice is the kinds of comparison predicates two or more containers are involved in. If we consider the XMark workload [Schmidt et al. 2002], we notice that, for example, the container corresponding to the path expression /site/people/person/@id, namely \text{C}\textunderscore id, is involved in an equality predicate with a constant in query \text{Q}\textunderscore 1 and in another equality predicate with the container corresponding to the path expression /site/closed\textunderscore auctions/closed\textunderscore auction/buyer/@person, namely \text{C}\textunderscore buyer, in query \text{Q}\textunderscore 8 of the workload. This means that the container \text{C}\textunderscore id is only involved in equality predicates, and that executing these predicates in the compressed domain is only possible if their compression algorithm supports it. This leads to choosing a data compression algorithm based on the predicates it supports in the compressed domain. For instance, Huffman compression [Huffman 1952] allows to evaluate equality predicates in the compressed domain,

1Here and in the remainder of the paper, by source model we mean the support structures used by the particular compression algorithm, e.g., a tree in the case of the Huffman algorithm.
but not inequality predicates. Other compression algorithms allow evaluating both equality and inequality predicates in the compressed domain. An example has been presented in [Antoshenkov 1997] (see also Section 4). More generally, XQueC considers a set of algorithms and their corresponding supported predicates, and chooses the algorithms for a particular data set in a way that mediates between several concerns: minimizing the occupancy of compressed containers, occupancy of the source model, and decompression time. The size of the total space of possibilities rules out exhaustive search, thus, XQueC applies several heuristics to choose compression algorithms optimizing these combined concerns.

Efficiency of query processing on compressed data XQueC’s fragmentation strategy provides fine-grained access to individual small data items. In turn, this selective access, and the usage of established node labelling schemes, provide the basis for efficient diverse query evaluation strategies in the compressed domains. In contrast, competitor systems, using more restricted (and compact) compressed formats, only allow for a single top-down evaluation strategy.

Thus, XQueC is the first queryable XML database management system capable of:

— compressing XML data and querying it as much as possible in the compressed domain;
— exploiting a storage model based on a fragmentation strategy that enables efficient query processing;
— make a cost-based choice of the compression algorithms, based on a query workload and a generic characterization of the properties of these algorithms.

These features allow XQueC to carefully balance compression performance aspects, such as the size of the compressed data or the incurred decompression time, and query execution performance, strongly determined by the available data access methods and query primitives. Each of these concerns can be given more or less weight, depending on the application needs.

The remainder of the paper is organized as follows. Section 2 presents an extensive study of related work and discusses the extensions presented in this contribution w.r.t. our previous published work [Arion et al. 2003], [Arion et al. 2004]. Section 3 illustrates the XQueC storage model. Section 4 presents the compression principles of XQueC and the cost model to make the compression choice effective and targeted to the data. Section 5 describes the query processing strategies built on the compressed storage structures. Section 6 presents the experimental assessment. Section 7 concludes the paper and discusses the future directions of this research.

2. RELATED WORK

Compression has always been a hot topic in relational databases and is all the more so in presence of XML databases. However, research on compressed XML databases cannot directly exploit the results achieved for relational data for two main reasons. First, the general interest towards compressing relational data has mostly been focused on numerical attributes, and poorly on string attributes, since the former were more frequent. For instance, in the TPC-H [Transaction processing performance council 1999] benchmark schema, only 26 of 61 attributes are strings. In contrast, in the popular XMark [Schmidt et al. 2002] benchmark for XML databases, 29 out of the 40 possible element content

---

2 By inequality predicates we mean those involving operators such as $>$, $<$, $\geq$, $\leq$, $!=$.
(leaf) nodes represent string values. (This figure does not include ID/IDREF attributes, which may also be counted as textual.)

Second, incorporating compression within relational databases has been complicated by the poor decompression performance of some algorithms, which may erase possible savings due to fewer disk accesses. This problem has widely been considered by past works on relational databases [Chen et al. 2000], [Westmann et al. 2000] and bitmaps [Amer-Yahia and Johnson 2000] and continues to hold for XML data.

We briefly introduce general-purpose compression algorithms in Section 2.1. Previous works having addressed text compression in non-XML databases are presented in Section 2.2. We then move to discussing XML compressors that do not support querying in Section 2.3. Finally, we present a comparison of existing queryable compressors for XML data (including the novelty of this paper with respect to our previous work) in Section 2.4.

2.1 General-purpose compression

Among the general string compression techniques, the Huffman algorithm [Huffman 1952] allows the optimal translation of a finite set of symbols into a target set of symbols holding the prefix property (no string in the target set is a prefix of any other string in the target set). The Hu-Tucker variant of this algorithm [Hu and Tucker 1971] produces order-preserving binary encodings. The Arithmetic compression algorithm [Witten 1987] translates a set of symbols into binary strings which are mapped in an interval $[0,1)$. The original Arithmetic compression algorithm was order-preserving, however, this feature is sacrificed for speed in other variants. However, both Hu-Tucker and Arithmetic translate one symbol at a time, and therefore are quite slow. Faster compression can be achieved by choosing substrings of the input to be used as substrings. This is more generally known as the dictionary approach. The Lempel-Ziv algorithm [Ziv and Lempel 1978] is dictionary-based, but it does not preserve order.

The Antoshenkov-Lomet-Murray (ALM) algorithm described in [Antoshenkov 1997], [Antoshenkov et al. 1996] is extensively used in XQueC for order-preserving compression, and is detailed in Section 4.1. ALM partitions the set of strings into intervals, and uses their common prefixes to achieve optimality. Thus, ALM can still achieve the benefits of dictionary-based encodings, without exhibiting the drawbacks of the prefix property among strings which leads to a blow-up of decompression times.

Finally, [Moura et al. 2000] proposes a variant of the Huffman algorithm (which we name throughout the paper as Extended Huffman), which allows phrase matching in the compressed domain. The queries supported include complex IR-style queries and regular expressions with wildcards, character ranges and complements, useful for text querying. However, the codes it produces are not order-preserving. In this paper, we address a more database-oriented subset of XQuery, where value selections and joins are made based on equality, inequality (cfr. footnote 2), and simple prefix predicates. A more detailed description of XQueC’s choices of compression algorithms is provided in Section 4.1.

2.2 Compression in non-XML databases

Several works have studied marrying compression and relational databases; among the most significant ones, we cite [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) the best compression granularity (field, block, tuple or file) file-level) from query performance viewpoint. [Goldstein et al. 1998]
proposes an encoding, called FOR (frame of reference), to compress numeric fixed-length fields, such as those used in the fact table of a star-schema in a data warehouse. However, we expect that an XML database contains textual data and not only numeric fields such as a data warehouse. [Westmann et al. 2000] discusses light-weight compression techniques oriented to field-level compression and [Greer 1999] uses both record-level and field-level encodings. However, field-level granularity compression does not translate directly to XML, due to the hierarchical data structure.

The impact of compression on the query processor and the query optimizer has been the subject of many past works in the relational domain. While Goldstein et al. [Goldstein et al. 1998] proposes a cost model that can be used to decompress data lazily as needed, [Westmann et al. 2000] discusses how to modify the relational query processor, the storage manager and the query optimizer in presence of field-level compression. [Chen et al. 2000] focuses on query optimization for compressed relational databases. A hierarchical dictionary encoding is used for strings. Furthermore, this work introduces transient decompression: intermediary results are decompressed in order to execute e.g. a join in the compressed domain, then re-compressed for the rest of the execution.

A novel lossy semantic compression algorithm oriented to relational data mining applications is presented in [Jagadish et al. 2004]. Finally, compression in a data warehouse setting is being applied in commercial systems, 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) will likely shift the interest of compression in data warehousing, from relational to XML.

2.3 Non-queryable compressors for XML databases

Among the non-queryable compressors for XML data, we cite XMill [Liefke and Suciu 2000], XMLZIP [XMLZIP 1999], and XML-PPM [Cheney 2001].

XMill [Liefke and Suciu 2000] is the first efficient XML-conscious compressor. It is based on the principle of separately compressing the data and the document structure. Data is split into containers in a default way (one container for each distinct element name) or in an arbitrary user-driven way. XMill compressed containers, first, using a customized semantic compressor, taking into account the particular data semantics, and then, re-compressing the whole with BZIP2 [BZIP2 2002]. However, human user expertise is needed in order to recommend the compression algorithm best suited for each container.

XMLZIP [XMLZIP 1999] and XML-PPM [Cheney 2001] are other XML compressors. XMLZIP compresses an XML document by clustering subtrees from the root to a depth d. However, this does not allow to exploit redundancies that may appear below this fixed level, thus some compression opportunities are lost. XML-PPM [Cheney 2001] implements ESAX, an extended SAX parser, which uses single bytes to encode tags. ESAX performance mainly depends on the the offline algorithm applied after the byte encoding, i.e., GZIP, BZIP or PPM.

2.4 Queryable compressors for XML databases

Our work most directly compares with queryable XML compression systems. We survey here the existing XML queryable compressors, that have been devised up to today.

The XGrind system [Tolani and Haritsa 2002] compresses XML by using an homomorphic encoding: an XGrind-compressed XML document is still an XML document, whose tags have been encoded by simple numbers, and whose textual content has been
compressed using the Huffman algorithm. XGrind’s query processor is an extended SAX parser, which can handle exact-match and prefix-match queries in the compressed domain. However, several operations are not supported by XGrind, such as non-equality selections in the compressed domain, joins, aggregations, nested queries, or new XML construction. Such operations occur in many XML query scenarios, as illustrated by XML benchmarks (e.g., all but the first two of the 20 queries in XMark [Schmidt et al. 2002]). Finally, XGrind only allows a fixed top-down query evaluation strategy.

XPRESS [Min et al. 2003] encodes whole paths into real numbers, and, like XGrind, compresses text leaves using the Huffman algorithm. The novelty of XPRESS lies in its path Reverse Arithmetic Encoding scheme, which represents each path by an interval of real numbers between 0.0 and 1.0. Queries supported in the compressed domain amount to exact/prefix queries and range queries with numerical values. Range queries with strings require decompression. Also, the navigation strategy is still top-down as the document structure is maintained by homomorphism.

The approach taken in [Buneman et al. 2003] compresses the structure of an XML document, based on a bisimulation relationship between the document and the compressed structure. However, going beyond regular dataguides, the compressed structure presented preserves enough information to support directly Core XPath [Buneman et al. 2003] node identification queries, a rich subset of XPath. Depending on the document considered, the compressed instance may sometimes be larger, other times smaller than the structural summary computed in XQueC. Their compressed instance may be also compared with the structural synopsis computed in [Polyzotis and Garofalakis 2002] and [Polyzotis and Garofalakis 2003] for estimating query selectivity. However, notice that this system is memory-based, and it does not produce a persistent compressed image of the data instance; furthermore, it does not compress leaf (text) nodes found in the documents. Thus, it does not compare directly to XQueC, whose explicit purpose is to support a large subset of the XQuery language (including, for instance, value-based selections and joins, and XML result constructions, among others.)

XQZip [Cheng and Ng 2004] uses a summary tree as the internal dataguide. Differently from dataguides though, this summary tree tends to merge subtrees containing the exact same set of paths, which may be too restrictive in some cases. For instance, if an optional element is present on a given path in the first tree, and is absent from that path in the second, the trees will not be eventually merged. Optional elements (corresponding to ? and * DTD markers) are quite frequent in XML documents [Mignet et al. 2003], [Schmidt et al. 2002]. For such documents, the XQZip summary tends to occupy an important size. Furthermore, XQZip applies GZIP compression to value blocks, which makes decompression mandatory even for equality predicates. The compression granules have a pre-defined length, empirically set at 1,000 records per block. At query processing time, XQZip tries to determine the minimum number of blocks to be decompressed, which may lead to decompress more than necessary. The queries addressable by XQZip in the uncompressed domain belong to an extended version of XPath, enriched with union and the grouping operator in the return step.

Table I outlines the major features of the mentioned systems. We compare them along several dimensions: structure and text compression, conformity to the homomorphic principle, value predicates directly answerable in the compressed domain, the entire language supported by the system (including through decompression) and finally, the set of available
strategies used when answering XML queries.

The specific advantages of XQueC consist, first, of its database-oriented design, which provides for efficient, scalable query processing, and second, of its capability to automatically determine the compression method best suited to a given XML dataset. These two aspects are combined in a single comprehensive architecture, which is functionally complete, yet extensible to include more compression algorithms than the ones currently supported. This comprehensive architecture allows XQueC to benefit from the performance advantages typical to uncompressed XML database systems, unlike its competitor systems, all the while having a smaller disk footprint thanks to compression techniques.

Comparison with our works. This extends our previous work presented in [Arion et al. 2003] and [Arion et al. 2004] in several ways. For what concerns the storage model, in the work presented here, we assign structural identifiers to XML elements, allowing XQueC to benefit from existing efficient XML pattern matching techniques. The query engine has been in great part re-built, and we have added new material describing it. We have improved our storage model, by a new container encoding which reduces furthermore the size of compressed containers. We have added a discussion on the impact of data and query workload updates on XQueC’s compression and execution model.

The cost model is better detailed and motivated. Besides joins and selections, the cost model also takes into account top-level projections and a richer set of algorithmic properties. Finally, we have improved over the greedy heuristic in [Arion et al. 2004], which is now replaced with a new set of cost-effective heuristics (i.e., $GH, G^2H$ and $CH$).

Finally, this paper includes a completely new set of experimental results describing XQueC performance and comparing it with other systems.

3. XQUEC’S QUERYABLE STORAGE MODEL FOR COMPRESSED XML DOCUMENTS

In this section, we present the storage model designed for our system with the purpose of still allowing queries in the compressed domain. The section is structured as follows. We start by introducing as running example the XMark benchmark documents, in Section 3.1. Then, Section 3.2 describes the layout of our storage model for XML document structure, and compressed values. Section 3.4 outlines the advantage of XQueC’s approach, based on a persistent store, over existing queryable XML compressors. Finally, Section 3.5 discusses the impact of updates on XQueC’s data organization.

3.1 Running example: XMark benchmark documents

Throughout this paper, we will use XMark [Schmidt et al. 2002] benchmark documents for illustrating our approach. Each document describes an auction site, including information...
about people buying and selling items, currently open auctions, closed auctions, etc. A sample XMark document is depicted at the top of Figure 1.

3.2 XQueC storage structures

We designed XQueC’s storage structures with the following goals: compactness, support for efficient XML query processing techniques, and lazy decompression (that is, the possibility of running queries mostly in the compressed domain).

The need for efficient query processing led us to adopt persistent node identifiers that enable selective data access; furthermore, we have adopted structural identifiers since they enable efficient query processing primitives [Al-Khalifa et al. 2002]. Although node identifiers may entail space overhead, they are of crucial importance to ensure scalability in any XML query processor. Moreover, to support lazy decompression, we have chosen small compression granules (the size of a data node). Finally, for efficiency reasons, we have adopted a highly fragmented storage, allowing the query processor to access only small parts of the data, as needed for a query. We now detail the individual storage structures resulting from this design.

XQueC splits an XML document into several data structures, depicted in Fig. 1.

Node name dictionary. XQueC uses a dictionary to encode the element and attribute names present in an XML document. Thus, if there are $N_t$ distinct names, XQueC assigns...
to each of them a bit string of length $\log_2(N_t)$. For example, the XMark documents use 92 distinct names, which are encoded on 7 bits.

**Structural identifiers.** XQueC assigns to each non-value XML node (element or attribute) a unique structural identifier, consisting of three integer values $[\text{pre}, \text{post}, \text{depth}]$ as in [Al-Khalifa et al. 2002]. These three numbers reflect the position of the XML node within the document. The $\text{pre}$ ($\text{post}$) number reflects the ordinal position of the element within the document, when traversing the document tree in preorder (postorder). Finally, the $\text{depth}$ number reflects the depth of the element in the XML document tree, i.e., the root element can be considered at depth 0, its children at depth 1, etc. The interest of such structural identifiers is well-known from [Al-Khalifa et al. 2002] and similar works such as [Grust 2002], [Halverson et al. 2003] and [Paparizos et al. 2003]: by examining two node identifiers, it is possible to directly infer the structural relationship among the two nodes, i.e., whether one is a parent or an ancestor of the other. Indeed, if $n_1$ and $n_2$ are two structural IDs, the node corresponding to $n_1$ is an ancestor of the node corresponding to $n_2$ if and only if $n_1.\text{pre} < n_2.\text{pre}$ and $n_1.\text{post} > n_2.\text{post}$. If, furthermore, $n_1.\text{depth} + 1 = n_2.\text{depth}$, then $n_1$ is more precisely the parent of $n_2$.

**Storing the XML tree structure.** For each path in the XML document tree, going from the root down to some node (whether root-to-leaf or root-to-intermediary node), XQueC stores a separate sequence consisting of all the structural IDs of the elements found on that path. Figure 1(c) depicts the sequences resulting from the paths /site, /site/people, /site/people/person, and /site/regions/asia/item, in the sample document above.

Thus, the structure of the XML document is stored as a set of logical sequences of structural IDs, each of which is associated to its given path.

**Remark: redundant depth field.** Note that once the structural identifiers are clustered by the path to which they belong, the depth field becomes redundant, as the depth is implicitly encoded in the path length. For example, all elements found on /site will have depth 1, elements on /site/people are at depth 2, those on /site/people/person are at depth 3, etc. Thus, the sequences in Fig. 1(c) use only a 2-tuple $[\text{pre}, \text{post}]$ to designate a structural ID.\(^3\)

At the physical level, to store the sequence of structural IDs in document order, we can use either a simple persistent sequence (very compact, but with bad performance in the case of document updates), or more generally an update-friendly persistent ordered storage structure (e.g., a B+-tree), which occupies more space, due to intermediary index pages and to the partially filled index pages. The choice of the best structure to use depends on the presence of updates in the query workload. We provide more details on the physical implementation we use in Section 6.1.1.

**Value containers.** All data values found under the same root-to-leaf path in the document are stored together into homogeneous containers. In our model, such values are either text values found inside XML elements, or values of XML attributes. In the absence of type information characterizing the XML document loaded, XQueC applies a simple type inference algorithm that attempts to classify the values on each path into one of the following types: string, integer, double. This simple type detection is applied when loading the document, in the spirit of XPRESS [Min et al. 2003]. A container is a sequence of container records, each one consisting of a compressed value, and a number representing

\(^3\)At query processing time, the depth field is also necessary and it is re-created, as described in Section 5.
the ordinal position of the value’s parent into its respective ID sequence. For instance, Fig. 1(e) shows some containers resulting from the sample XMark document. We denote the result of compressing a string \( s \) as \( c(s) \). In the first compressed container, each compressed value is paired with an integer corresponding to the index of the respective parent element, within the corresponding ID sequence shown at left.

ID sequences and containers can be seen as an aggressive vertical partitioning model, reminiscent of traditional distributed database techniques [Ozsu and Valduriez 1999]. Such partitioning provides direct access to various parts of the document, namely, at ID sequence or container granularity level. This choice provides interesting query evaluation strategies and leads to good query performance (see Section 6).

Structure summary. The loader also constructs, as a redundant access support structure, a structure summary representing all possible paths in the document.

A structure summary is obtained from the XML document \( d \), by constructing:

— one root node corresponding to \( d \)’s root, sharing the same label;
— for each node \( n_d \in d \), let \( s(n_d) \) be the structure summary node corresponding to \( n_d \). If \( n_d \) has at least a child \( n_d' \) labeled \( l \) in \( d \), then \( s(n_d) \) will have exactly one child labeled \( l \), and that child represents in the structure summary all \( l \)-labeled children of \( n_d \);
— for each node \( n_d \in d \), let \( s(n_d) \) be the structure summary node corresponding to \( n_d \). If \( n_d \) has some text children, then \( s(n_d) \) will have exactly one child labeled \#text, and that child represents in the structure summary all text children of \( n_d \). Similarly, if \( n_d \) has an attribute labeled \( a \), then \( s(n_d) \) will have exactly one child labeled @\( a \), representing in the structure summary the \( a \) attribute of \( n_d \).

Fig. 1(b) depicts the structure summary for the sample document in Fig. 1(a). Clearly, for every path \( /a_1/a_2/\ldots/a_k \) in the document \( d \), there is exactly one node reachable in the structure summary of \( d \) by the same path. Thus, we establish a bijection between paths in an XML document and nodes in the structure summary. Similarly, each leaf node in the structure summary (labeled \#text in the case of text elements, and @\( a \) in the case of attributes) corresponds to a container of compressed values.

Notation. We assign integer numbers 1, 2, 3, \ldots to the nodes of the structure summary. By extension, we use these numbers to refer to the corresponding paths, and to the corresponding ID sequence. Similarly, we refer to containers by appending \#text or @\( a \) after a path number. For instance, Fig. 1(d) shows the logical layout of the ID sequences from Fig. 1(c): each sequence is uniquely identified by the number of a summary node. Similarly, Fig. 1(f) depicts containers, identified by the number of their parent and by \#text or an attribute name.

The structure summary is used for query optimization purposes; we describe this in more details in Section 5.3. The summary is typically very small (we illustrate this in Section 6), thus, it does not significantly impact data compression.

When loading a document, other indexes and/or statistics can be created, either on the value containers, or on the structure tree. As a side-effect of document loading, our XQueC prototype gathers a simple set of fan-out and cardinality statistics as follows. Let \( p \) be a path in the structure summary of the form \( /a_1/a_2/\ldots/a_k \).

XQueC attaches to the structure summary node of path \( p \) the following statistics:

— \( N_p \): the number of elements found on the path \( /a_1/a_2/\ldots/a_k \), that is, the size of the
sequence of identifiers found on this path.
— For each path \( p = /a_1/a_2/ \ldots /a_k/a_{k+1} \) in the structure summary, XQueC attaches to \( p \) the
numbers \( m_{p,p'} \) and \( M_{p,p'} \): the minimum, resp. maximum number of children on path \( p' \)
of the XML elements on the path \( p \).

These statistics are illustrated in Fig. 1(g). XQueC loads an XML document in a single
pass, using a customized SAX [XML-SAX 2000] parser. The algorithm is similar to the
one described in [Aboulnaga et al. 2001].

**Notation.** For a given document, we denote by \( N \) its size, \( h \) its height, and by \( N_S \) the
number of nodes in its structure summary.

Using these notations, the loading algorithm runs in time linear with \( N \), using
\( O(h + N_{PS}) \) memory for the stack and structure summary.

**XQueC compression factor.** The compression factor is commonly defined as:

\[
 cf = 1 - (cs/os) \tag{1}
\]

where \( cs \) and \( os \) are the sizes of the compressed and original documents, respectively.

While the abovementioned systems transform an XML document into one single com-
pressed file, XQueC transforms it into several storage structures. Thus, the size of the
compressed document, as produced by XQueC, which we denote by \( cs_{XQueC} \), deserves a
more detailed explanation. We have:

\[
 cs_{XQueC} = ccont_{XQueC} + cstruct_{XQueC} + aux_{XQueC} \tag{2}
\]

where: \( ccont_{XQueC} \) is the total size of all the compressed containers, \( cstruct_{XQueC} \) is
the total size of all ID sequences, and \( aux_{XQueC} \) is the total size of the path summary and
of all compression dictionaries stored by XQueC.

Let \( N \) be the number of elements in the document. We need \( \lceil \log_2(N) \rceil \) bits to represent
an element number, thus \( 2 \times \lceil \log_2(N) \rceil \) bits to represent a [pre, post] element ID. Thus:

\[
 cstruct_{XQueC} = N \times 2 \times \lceil \log_2(N) \rceil \tag{3}
\]

The size occupied by a compressed container \( c \) reflects the size of the compressed values
themselves, plus the size needed to store the integers referring to the parent element in its
ID sequence, denoted \( IDS_{eq} \). Let \( |c| \) denote the number of entries in a container, and let
\(|x|\) be the number of bits occupied by \( x \) if \( x \) is a simple atomic value. Furthermore, let \( c_i \)
be the \( i \)-th compressed value in the container \( c \). We have:

\[
 ccont_{XQueC} = \sum_c \left( |c| \times \lceil \log_2(|IDS_{eq}|) \rceil + \sum_{i=1, \ldots, |c|} |c_i| \right) \tag{4}
\]

Finally, \( aux_{XQueC} \) can be detailed as:

\[
 aux_{XQueC} = \sum_{n \in PS} \left( |tag(n)| + 16 + 2 \times \lfloor int \rfloor \right) + \sum_{p \in partitions} |dict(p)| \tag{5}
\]

In (5), the first term represents the space needed for the storage of the path summary.
For each summary node, we store its tag, the \( M_{x,y} \) and \( m_{x,y} \) numbers, and use two bytes
to store the record length for this node.
The second term in (5) corresponds to the dictionary size; this term depends on the compression algorithm used, and also on the data set. Each dictionary is stored only once for the container group to which it has been applied.

3.3 XQueC's storage model vs. other XML storage and indexing structures

We briefly discuss here how XQueC's storage model compares with storage models used in previous XML data management systems.

The usage of a Dataguide [Goldman and Widom 1997] has been pioneered in the Lore system, and subsequently as an indexing basis [McHugh and Widom 1999]. In XQueC, the summary is part of the storage itself (indeed, it constitutes the data catalog) and is required for query processing, whereas it was used as an optional index structure in [McHugh and Widom 1999]. Due to its distinct usage and required aspect, we prefer using the term "structure summary" instead of "strong Dataguide", to avoid confusion. 1-indexes [Milo and Suciu 1999] can be seen as "generalized Dataguides" for XML, in that they group together nodes accessible by general paths, including simple paths (as the Dataguides do) but also more general paths, for instance, featuring // . Our structure summary closely resembles a dataguide.

Dataguides, annotated with the statistics described above, have been used also in [Aboul-naga et al. 2001] for estimating path query cardinalities. Furthermore, [Aboulnaga et al. 2001] studies ways of pruning such a dataguide, to make sure it fits in memory. In XQueC, all the summary is necessary for query processing, thus pruning is ruled out. Part of our ongoing work focuses on handling it in a streaming fashion, to avoid memory problems in the case of large summaries.

Structural identifiers have recently grown very popular. They are used, for instance, within the Timber, Niagara and Natix systems [Jagadish et al. 2002; Fiebig et al. 2002; Halverson et al. 2003]. The storage models of these systems consist of a persistent tree, complemented by a tag-partitioned index of structural identifiers. Thus, for a given tag such as name, the tag-partitioned index allows to retrieve all structural identifiers of elements having that tag. XQueC's model is different in several ways.

— First, we do not use a persistent tree as storage; this allows us to reduce the disk footprint significantly.

— Second, XQueC's path partitioning dispenses it from having to store the tags (even dictionary-encoded tags) multiple times. XQueC only stores one (dictionary-encoded) tag for each node in the structure summary. In contrast, tag-based indexes used in [Jagadish et al. 2002; Fiebig et al. 2002; Halverson et al. 2003] have to store the tags as keys in the B-trees.

— Third, if persistent sequences are adopted as physical data structures for ID sequences and containers, then the space overhead incurred by B-trees is avoided.

— Fourth, path partitioning dispenses XQueC from having to store the depth field in structural identifiers, which may spare 1 byte per identifier in some cases.

These differences allow XQueC (even if values are not compressed) to use slightly less than the original file size in order to store it (notice that we include in this size all the structural identifiers we introduce). Of course, value compression further diminishes this footprint, depending on the data set. In contrast, systems like Timber take about four times the size of the original file in order to store it, as reported in [Halverson et al. 2003].
Besides being compact, XQueC’s storage also allows for more efficient query processing than the models mentioned above. We detail this in Section 5.4, after having introduced our query processing model. However, since XQueC’s storage does not include a persistent tree, reconstructing complex data trees is an issue. We address this also in Section 5.

3.4 Discussion: persistent store vs. in-memory queryable compressors

Data fragmentation as performed in XQueC allows direct data access and provides for a variety of query evaluation strategies, as in regular XML data management systems. In contrast, homomorphic compressors such as XGrind [Tolani and Haritsa 2002] and XPRESS [Min et al. 2003] do not use a persistent store; query evaluation is done in a single SAX parsing step. This has two restrictive consequences.

First, it may entail the need to traverse the whole document in order to answer queries including //; in contrast, a fragmented storage (such as the XQueC’s) allows direct access to interesting parts of the document.

Second, XGrind and XPRESS are restricted to those queries that can be evaluated in a single SAX parsing step; this rules out XQueries containing joins, reconstructions, nesting etc., which XQueC is able to handle. Furthermore, SAX-based evaluation requires building an automaton in memory, which can get quite large for complex XPath queries. The queries studied in the XGrind [Tolani and Haritsa 2002] and XPRESS [Min et al. 2003] papers indeed feature few path predicates, and none at arbitrary depth (of the form //a,[//a,]). The streaming evaluation of the latter class of queries is more complex, since the automaton may be large.

In contrast, the XQueC storage model hardcodes all paths into the storage, providing selective access to the data for query processing; furthermore, XQueC’s data access operators being pipelined, they have negligible memory consumption (see Section 5.1). Consider as example the following query, Q_{14} in the XMark benchmark:

```xml
for $i in document("auction.xml")/site//item
where contains ($i/description,"gold")
return $i/name/text()
```

This query could be handled by XGrind and XPRESS, but these would have to traverse all the document to find relevant items, as shown in Fig. 2.

In XQueC, the compressor has already shredded the data and accessibility to these data from the structure summary allows to save the parsing and loading times. Thus, in XQueC the structure summary is parsed (not all the structure tree), then the involved containers are directly accessed (or alternatively their selected single items) and loaded into main-memory. More precisely, as shown in Fig. 2, once the structure summary leads to the containers C_1, C_2 and C_3, only these (or part of them) need to be fetched in memory.

3.5 Impact of updates on XQueC’s data organization

In this section, we discuss the impact of updates to the XML document on the persistent storage format of XQueC. This impact has several facets.

First, independently of compression, XML updates tend to break the [pre, post, depth] structural identifiers used in XQueC. To better support updates, more update-friendly node labeling schemes [Li and Moon 2001; O’Neil et al. 2004] could be used. A simple solution used in Timber [Jagadish et al. 2002] is to use real numbers instead of integers for pre, post and depth. All these solutions increase the size of the persistent IDs. However, this
issue arises for all persistent XML stores using structural identifiers, and there are well-understood limitations on how small identifiers can get, while still allowing updates [Cohen et al. 2002]. In XQueC, since our focus is on compression, we naturally favoured storage compactness, at the expense of updates.

Second, again independently of compression, the insertion of new values between existing ones in containers, or the insertion of new IDs within an ID sequence are quite time-consuming if disk-based sequences are used as physical storage structures. Depending on the frequency and type of updates, B-trees may become a preferrable physical structure to back up containers and ID sequences. We have implemented and experimented with both B-trees and persistent sequences; however, since our interest was not on updates, the default storage is sequence-based.

Third, inserts may lead to creating new paths in the document. The structure summary, together with the query shape, allow to understand on which paths is some new data inserted, and check if these paths existed already. For instance, consider the following update directive, expressed in the syntax of [Sur et al. 2004]:

```
update
...for $x in //asia//item
...where $x/initial>100
...insert <note> I think this item is <keyword>nice</keyword></note> as last into $x
```

The structure summary allows to infer that $x$ can only be on the path /site/regions/asia/item. From the update directive syntax, we understand that the new data is to be added under this path. Then, from the XML fragment describing the child to be added to $x$, we understand that the new data should go into:

— The ID sequences pertaining to the paths /site/regions/asia/item/note and /site/regions/asia/item/note/keyword.

— The containers storing (compressed) values from the paths /site/regions/asia/item/note/#text and /site/regions/asia/item/note/keyword/#text.

Thus, one can check for the presence of these structures, and create them if needed.

Fourth, new data inserted in the document must be compressed. New values to be added to an existing container can be compressed using the container’s source model. When a new container is created and there is no indication of it in any query in the workload, then a simple conservative choice (considering the container as a singleton) may be made. We discuss the impact of changes in the set of values of a container or, alternatively, the addition of a new query in the workload in Section 4.2.
4. CHOICES OF A COMPRESSION CONFIGURATION

In this section, we present the principles adopted in XQueC to choose the compression algorithms for a given set of containers. In principle, one could use the same compression algorithm for the entire set of containers. However, each compression algorithm brings along some associated properties and associated costs. For instance, depending on the algorithm chosen, only some value comparisons are feasible in the compressed domain. We will discuss costs, and show that, in order to obtain a suitable compression configuration, a query workload may play a crucial role. A query workload is not difficult to build [Roy et al. 2000; Bohannon et al. 2002], and can be beneficial to the choice of a compression configuration. In particular, we illustrate here the necessity of having a cost model and suitable heuristics deciding which algorithms are to be applied to a given set of containers.

Section 4.1 introduces some families of compression algorithms that may constitute the input of the cost model, and discusses their properties (the model is general to allow as input any set of compression algorithms). In Section 4.2 we illustrate the cost model and some heuristics to compute suitable compression configurations.

4.1 Compression algorithms as input to the cost model

As a first observation, within the XQueC cost model, we consider compression algorithms exhibiting useful properties for string comparisons. Indeed, strings are much more frequent in XML documents than numerical values and as a consequence, string compression, when effective, can truly reduce the space occupancy of XML documents. Moreover, since we are building a queryable compressor, we would be interested to keep the querying time reasonably low. In particular, we would like to choose algorithms that, besides a good compression ratio, also allow selections and 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 here 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, Huffman [Huffman 1952] supports both equality selections and equality joins in the compressed domain. Same holds for ALM [Antoshenkov 1997], extended Huffman [Moura et al. 2000], Arithmetic [Witten 1987] and Hu-Tucker [Hu and Tucker 1971]. This property does not hold for those algorithms, such as adaptive Huffman coding, and FGK [Lelewer and Hirschberg 1987], that compute the source model on-the-fly during compression: this implies that the same string might not be compressed the same way when appearing at different positions in the input.

— **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 (“c1 like c2*”) can be evaluated in the compressed domain. This property holds for the Huffman algorithm, not for ALM.

— **regular expression-preserving compressors**: these algorithms allow to evaluate 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 to determine inequality selections, as these can be equivalently expressed as regular expression selections. Note also that the ability to match a regular expression in the compressed domain does not ensure the possibility of testing if “c₁ like c₂” without decompressing c₁ or c₂. In other words, with these algorithms, we cannot make regular expression joins. An example of an algorithm supporting regular expression selections (but not joins) is Extended Huffman.

Given that there are several compression algorithms, each with various interesting properties for querying, the choice of which algorithm(s) to use to compress XML data is other than trivial. In XQueC, a cost model engenders the final choice of which algorithm(s) are the most suitable for compressing the data to be queried afterwards (Section 4.2). The cost model is designed to work with any number/kind of compression algorithms as input. In fact, every new algorithm can be described in terms of the algorithmic properties considered, that are formalized in the sequel. We implemented ALM and Huffman as a proof of concept in XQueC, and therefore we illustrate our cost model analysis based solely on their properties, that turn to be interesting for plain XQuery [XQUE 2004]; other properties, such as those exhibited by regular expression-preserving algorithms, can be easily included, for instance in order to optimize XQuery full-text queries [USE 2003] evaluation. The Huffman algorithm compresses one character at a time and achieves asymptotically optimal compression; it is relatively fast, its compression dictionary is typically small, and it supports equality comparisons in the compressed domain. For order-preserving compression, we preferred ALM over the Arithmetic and Hu-Tucker algorithms, as dictionary-based encoding has demonstrated its effectiveness w.r.t. other non-dictionary approaches [Moffat and Zobel 1992], and ALM outperformed Hu-Tucker [Antoshenkov et al. 1996].

4.2 Compression choices
Since the containers are typically a large share of XML data, carefully choosing the compression algorithms suitable for individual/multiple containers is a relevant feature of an XML compressor [Liefke and Suciu 2000]. When the XML compressor needs to be queryable, however, this choice becomes more crucial, due to the fact that the containers are also involved into comparisons. Thereafter, to make the right compression choice, it is needed to look at the comparisons in a given query workload. Query workloads have been already successfully employed in several performance studies, from multi-query optimization to XML-to-relational mappings [Roy et al. 2000; Bohannon et al. 2002]. Even when one compression algorithm is used, the cost model comes at hand to properly associate containers exhibiting data commonalities and optimize the compression costs.

But how do we know that a compression algorithm is suitable for a container or a set of containers? Many algorithms can be used, but we would like to choose one with nice properties. For instance, the decompression time of an algorithm strongly influences the query response times over data compressed with that algorithm. Also, the compression ratio achieved by an algorithm on a container depends both on the algorithm and on the nature of data in the container. In addition, a container can be compressed individually or along with other containers. In the latter case, a group of containers share the same source model, as built by the compression algorithm. This choice might be convenient, e.g., when two containers have similar values. Therefore, the occupancy of the source
model matters, as much as the occupancy of containers themselves, and the decompression time; combining these three factors makes the choice even more challenging.

In order to illustrate the impact of compression choices, consider a very simple case with two containers compressed using the Huffman algorithm; each of the two containers contains strings composed over an alphabet of two symbols, and the two alphabets are disjoint. In this situation, if two separate source models are built for the two containers, the containers are encoded with one bit per symbol. If instead a single source model is built for both, this requires two bits per symbol, degrading the compression ratio. Keeping the two containers separated may require decompressing them, when a comparison not supported by the compression algorithm must be evaluated. The scenario gets more complex when choosing among more compression algorithms, thus the choice of the algorithms to use must be carefully balanced. To this purpose, XQueC employs the cost model described in the next section.

4.2.1 Cost model: appreciating the quality of a compression configuration.

Notation and background. The XQueC cost model works on a set of textual containers, \( \mathcal{C} \), a set of available compression algorithms, \( \mathcal{A} \), a query workload, \( \mathcal{W} \), and a set \( \mathcal{L} \) of algorithmic properties, i.e., the kinds of comparisons considered, denoted by labels (e.g., \( \text{eq}_j \) for equality joins, \( \text{ineq}_s \) for inequality selections, etc.). A compression configuration \( s = \langle P, \text{alg} \rangle \) for \( \mathcal{C} \) consists of a partition \( P \) of \( \mathcal{C} \)'s elements, and a function \( \text{alg} : P \rightarrow \mathcal{A} \) associating a compression algorithm to each set in \( P \) (see Table II).

A configuration \( s \) dictates that the containers in each set \( p \in P \) will be compressed using \( \text{alg}(p) \) and a common shared source model. A suitable cost function, when evaluated on a configuration \( s \), must reflect various costs: the cost of decompressions needed to evaluate comparisons and projections in \( \mathcal{W} \); the compression ratios of the different algorithms employable; the cost of storing the source models used by the different algorithms.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{C} )</td>
<td>a set of textual containers</td>
</tr>
<tr>
<td>( \mathcal{A} )</td>
<td>a set of compression algorithms</td>
</tr>
<tr>
<td>( \mathcal{W} )</td>
<td>a query workload</td>
</tr>
<tr>
<td>( P )</td>
<td>a partition of ( \mathcal{C} )</td>
</tr>
<tr>
<td>( s )</td>
<td>a set in ( P )</td>
</tr>
<tr>
<td>( \mathcal{L} )</td>
<td>kinds of comparisons considered</td>
</tr>
<tr>
<td>( \text{alg} )</td>
<td>a function ( P \rightarrow \mathcal{A} )</td>
</tr>
<tr>
<td>( l )</td>
<td>a kind of comparison</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>an algorithm in ( \mathcal{A} )</td>
</tr>
<tr>
<td>( F )</td>
<td>a similarity matrix</td>
</tr>
<tr>
<td>( F_p )</td>
<td>similarity matrix over the containers in ( p )</td>
</tr>
<tr>
<td>( c_{\text{d}}(F) )</td>
<td>cost of decompressing a symbol</td>
</tr>
<tr>
<td>( c_{\text{s}}(F) )</td>
<td>cost of storing a compressed symbol</td>
</tr>
<tr>
<td>( c_{\text{a}}(F) )</td>
<td>cost of auxiliary structures</td>
</tr>
<tr>
<td>( \text{cmp}_{\mathcal{W}} )</td>
<td>set representing comparisons in ( \mathcal{W} )</td>
</tr>
<tr>
<td>( \text{proj}_{\mathcal{W}} )</td>
<td>set representing top-level projections in ( \mathcal{W} )</td>
</tr>
<tr>
<td>( d(s, i, j, l) )</td>
<td>decompression cost due to a comparison of kind ( l ) between containers ( c_i ) and ( c_j )</td>
</tr>
<tr>
<td>( d'(q, s, i) )</td>
<td>decompression cost due to a projection in query ( q ) on container ( c_i )</td>
</tr>
</tbody>
</table>

Table II. Summary of symbols used in the cost model.
Characterization of Compression Algorithms. Each algorithm \( a \in \mathcal{A} \) is denoted by a tuple \( \langle a[c_d(F)], a[c_s(F)], a[c_a(F)], a[L] \rangle \), with:

- a similarity matrix \( F \): a symmetric matrix whose generic element \( F_{i,j} \), with \( 0 \leq F_{i,j} \leq 1 \), is the normalized similarity degree between \( c_i \) and \( c_j \).
- the decompression cost \( a[c_d(F)] \): a function estimating the cost of retrieving an uncompressed symbol from its compressed representation using algorithm \( a \).
- the storage cost \( a[c_s(F)] \): a function estimating the average cost of storing the compressed representation of a symbol using \( a \).
- the source model storage cost \( a[c_a(F, \sigma)] \): a function estimating the cost of storing the auxiliary structures needed to represent the source model of a set of containers sized \( \sigma \) using \( a \).
- the algorithmic properties \( a[L] \): a set containing 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 into the containers compressed together, i.e., on the similarity among them (see the example in the previous section). Observe also that the source model storage cost is not specific to each container symbol, but it refers to the storage of an entire source model. This is due to the fact that the overhead of storing the source model is seldom linear w.r.t. containers size [Moura et al. 2000].

An eligible similarity matrix. A particular characterization of \( F \) has been employed in our experiments (see Section 6); the characterization of \( F \) and \( c_x(F) \) for compression algorithms cannot be exhaustive here as it is outside the scope of this paper. To build matrix \( F \), we chose the Cosine similarity function as it is able to look at the relative symbol frequency distribution, without taking into account the absolute occurrences of symbols, when combined with a symbol-wise vector representation of container contents. The cosine similarity is defined as the cosine of the angle between the vectors that represent the containers. More formally, we first define the signature of a container as the number of occurrences of a fixed set of symbols \( \Sigma \) (composed of characters of the western alphabet plus some punctuation). Thus, the signature of a container \( c \) can be defined as a function \( \tau : \Sigma \to \mathbb{N} \). The cosine similarity between two containers \( c_i \) and \( c_j \) is therefore defined as follows:

\[
F_{i,j} = \frac{\sum_{x \in \Sigma} \tau_i(x) \cdot \tau_j(x)}{\sqrt{\sum_{x \in \Sigma} \tau_i^2(x)} \cdot \sqrt{\sum_{x \in \Sigma} \tau_j^2(x)}}
\]

Storage costs. The containers storage cost for each set \( p \in P \) is computed by multiplying the overall number of symbols in \( p \) by the storage cost incurred by the algorithm that \( s \) associates with \( p \); the source model structures storage costs is instead independent of the number of symbols. Such costs are influenced by the similarity among the containers in \( p \), so they are evaluated on the projection of \( F_C \) w.r.t. the containers in \( p \) (denoted as \( F_p \)). Thus, the containers storage cost is

\[
\text{sec}(s) = \sum_{p \in P} (\text{alg}(p)[c_s(F_p)]) \cdot \sum_{c \in p} |c|
\]

where \( |c| \) denotes the total number of symbols appearing in container \( c \), and the source model structures storage cost is
Decompression cost. We begin by considering two sets, namely $\text{cmp}_W$ and $\text{proj}_W$, that reflect selections and joins among containers, and top-level projections in $W$. $\text{cmp}_W$ is a set of tuples of the form $(q, i, j, l)$, where $q \in W$ is a workload query, $i \in \{1, \ldots, |\mathcal{C}|\}$, $j \in \{1, \ldots, |\mathcal{C}| + 1\}$ are container indexes (index $|\mathcal{C}| + 1$ represents constant values for selections), and $l \in \mathcal{L}$; each tuple denotes a comparison of kind $l$ in $q \in W$ between containers $c_i$ and $c_j$. $\text{proj}_W$ is instead a set of tuples of the form $(q, i)$, where $q \in W$ is a workload query, and $i \in \{1, \ldots, |\mathcal{C}|\}$ is a container index; each tuple in $\text{proj}_W$ denotes a projection on container $c_i$ in $q \in W$.

The decompression cost is evaluated by summing up the costs associated with both comparisons and projections in $W$. 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 needed kind of comparisons in the compressed domain. A non-zero decompression cost occurs instead when one of the following conditions holds:

1. $c_i$ and $c_j$ are compressed using different algorithms;
2. $c_i$ and $c_j$ are compressed using the same algorithm but different source models;
3. $c_i$ and $c_j$ are compressed using the same algorithm and the same source model, but the algorithm does not support the needed 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 needed kind of selection in the compressed domain. In such a case, the constant value can be compressed using $c_i$’s source model and the selection evaluated directly in the compressed domain. To formalize this, we define a function $d$ that, given a compression configuration, calculates the cost of decompressing pairs of containers or single containers (if involved in selections).

**Definition 4.1.** Let $s$ be a compression configuration, $i \in \{1, \ldots, |\mathcal{C}|\}$ and $j \in \{1, \ldots, |\mathcal{C}| + 1\}$ container indexes, and $l \in \mathcal{L}$. $d(s, i, j, l)$ is defined as follows:

$$d(s, i, j, l) = \begin{cases} |c_i| \cdot \text{alg}(p)[c_d(F_{p'})] + |c_j| \cdot \text{alg}(p')[c_d(F_{p''})] & \text{if } j \neq |\mathcal{C}| + 1, c_i \in p', c_j \in p'', (p' \neq p'' \text{ or } l \notin \text{alg}(p')[|\mathcal{C}|]), \\ |c_i| \cdot \text{alg}(p)[c_d(F_p)] & \text{if } j = |\mathcal{C}| + 1, c_i \in p, l \notin \text{alg}(p')[|\mathcal{C}|], \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, top-level projections in $W$ incur a non-zero decompression cost only if there is no comparison in the same query that already requires decompression of the projected containers. To capture this, we define a function $d'$ that, given a query in $W$ and a compression configuration, calculates the decompression cost associated with a projection over a container.

**Definition 4.2.** Let $q \in W$ be a workload query, $s$ a compression configuration, and $i \in \{1, \ldots, |\mathcal{C}|\}$ a container index. $d'(q, s, i)$ is defined as follows:

$$d'(q, s, i) = \begin{cases} |c_i| \cdot \text{alg}(p)[c_d(F_p)] & \text{if } j = |\mathcal{C}| + 1, c_i \in p, l \notin \text{alg}(p')[|\mathcal{C}|], \\ 0 & \text{otherwise.} \end{cases}$$
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 $W$. Note that the costs associated to joins are properly adjusted to take into account that, if a join involves twice a certain container, it is obviously decompressed once.

**Definition 4.3.** Let $s$ be a compression configuration. $\text{decomp}_W(s)$ is defined as follows:

$$\text{decomp}_W(s) = \sum_{(q, i, j, l) \in \text{cmp}_W} d(s, i, j, l) \cdot \text{adj}(i, j) + \sum_{(q, i) \in \text{proj}_W} d'(q, s, i)$$

where $\text{adj}(i, j) = \frac{1}{2}$ if $i = j$, and $\text{adj}(i, j) = 1$ if $i \neq j$.

Observe that, should the queries in $W$ have different frequencies of occurrence, the decompression cost formula could be enhanced by viewing $\text{cmp}_W$ and $\text{proj}_W$ as bags instead of sets.

**Cost Function.** The overall cost of a configuration $s$ w.r.t. a workload $W$ is calculated as a weighted mean of the costs seen above:

$$\text{cost}(s, W) = \alpha \cdot \text{decomp}_W(s) + \beta \cdot \text{sc}(s) + \gamma \cdot \text{scm}(s)$$

where $\alpha$, $\beta$, and $\gamma$, with $\alpha + \beta + \gamma = 1$, are suitable cost weights that measure the relative importance of the various components involved. It can be shown that the time complexity of function $\text{cost}$ is $O(|W| \cdot |C|^2)$ under the assumption that the time complexity of functions $c_x(F)$ is $O(|F|)$.

**Impact of updates on the cost model.** Updates may occur in several places, including the stored data, the set of employable algorithms, and the query workload. All these cases impact the cost model choices. A change in a container’s content is reflected in both the similarity of that container w.r.t. the others, and in the number of symbols considered within the components of the cost function. The addition (deletion) of compression algorithms adds new characterizing tuples, that in turn involve all the components of the cost function. Finally, the addition (deletion) of queries in the workload (for instance due to new trends in users preferences) affects the component $\text{decomp}_W$ of the cost function. In any case, the complexity of computing the cost of a configuration remains the same, while the effort needed to recompute a suitable compression configuration depends on the particular algorithm adopted.

However, changes in the set of algorithms employed and in the query workload happen very rarely to need a recomputation. Indeed, typically the compression algorithms are decided before starting a long query session and remain stable for a while, and the same happens for the query workload. Conversely, updates on data may be more frequent; however, small variations, especially in the case of large containers, do not necessarily lead
to recompute the compression configuration from scratch. Indeed, adding or replacing few items does not deeply impact the cost model functions, even if it may still change the container size. In conclusion, although in principle a recomputation is needed each time a variation occurs, in practice we can just check that $c_d$, $c_a$ and $c_s$ are not significantly altered and live with the current compression choices. Another suitable choice in presence of more frequent updates would be that of periodically recomputing the compression configuration. For such a reason, we need fast algorithms to be able to compute suitable configurations in a reasonable time. This is discussed in the following.

4.2.2 Optimizing compression choices. The problem we deal with is that of finding, given a query workload $W$, a set of containers $C$, and a set of compression algorithms $A$, the configuration incurring the minimum cost. Observe that the explored search space may tremendously grow. Being $P$ the set of possible partitions of $C$, $|P|$ is the Bell number $B_{|C|}$, which is exponential with $|C|$. Moreover, for each possible partition $P \in P$, there are $|A|^{|P|}$ ways of assigning a compression algorithm to each set in $P$. Therefore, the total size of the search space is in theory $\sum_{P \in P} |A|^{|P|}$.

Exhaustively exploring the entire search space is quite prohibitive in practice. Moreover, in the presence of updates, it might be needed to perform periodic re-computations of the compression configurations employed. We have designed some simple and fast heuristics aiming at computing suitable compression configurations, while still taking acceptable time: a greedy heuristic (GH) which makes locally-optimal merges starting from a reasonable initial configuration; a group-based greedy heuristic (G$^2$H) that adds a preliminary step to GH grouping together presumably similar containers; a clustering-based heuristic (CH) that employs a cost-based distance measure combined with a classical hierarchical clustering algorithm.

Greedy heuristic. The first heuristic we devised is a greedy one. The search starts with an initial configuration $s_0 = < P_0, alg_0 >$, and tries to head towards better solutions by hierarchically combining the sets in the partition. The main idea here is that of considering each comparison in $W$ to enhance the current configuration by reconsidering the sets to which the containers involved in the comparison belong.

More formally, $s_0$ is built as follows:

1. $W' \leftarrow W$.
2. Repeat
   (a) Choose $i$ and $j$ such that containers $c_i$ and $c_j$ have the maximum number of comparisons in $W'$ between them.
   (b) Erase all comparisons in $W'$ involving both $c_i$ and $c_j$.
   (c) If no set in $P_0$ already contains $c_i$ or $c_j$, add the set $p^n = \{c_i, c_j\}$ to $P_0$ and erase all comparisons in $W$ involving both $c_i$ and $c_j$.
   (d) If at the previous step a new set is added to $P_0$, make $alg_0$ associate with $p^n$ the algorithm capable of covering the maximum number of comparisons between $c_i$ and $c_j$ in the compressed domain. If different algorithms reach the same number, $alg_0$ is the one that minimizes the expression $\alpha \cdot \text{alg}_0(p^n)[|c_d(F_{p^n})|] + \beta \cdot \text{alg}_0(p^n)[|c_a(F_{p^n})|] + (1 - \alpha - \beta) \cdot \text{alg}_0(p^n)[|c_s(F_{p^n})|]$.
   until $W' = \emptyset$.
3. Add to $P_0$ a singleton for each container not already belonging to a set in $P_0$. Assign to the singleton an algorithm chosen as at point (2d).
A move from \( s_k = < P_k, alg_k > \) to \( s_{k+1} = < P_{k+1}, alg_{k+1} > \) proceeds as follows:

1. Pick the comparison \( pred \in \mathcal{W} \) having the maximum number of occurrences. Let \( c_i \in p' \) and \( c_j \in p'' \) be the containers involved in \( pred \).

2. Build the configurations \( s_{k1} = < P_{k1}, alg_{k1} >, \ldots, s_{k|A|} = < P_{k|A|}, alg_{k|A|} > \) where \( P_{k1} \) is obtained from \( P_k \) by replacing \( p' \) and \( p'' \) with their union \( p^u \). The different assignment functions coincide with \( alg_k \) for all the sets in \( P_{k1} \), but each of them assigns to \( p^u \) a different algorithm.

3. Compare the costs of the newly built configurations with the current configuration cost, and let \( s_{k+1} \) be the one with the minimum cost.

4. Erase from \( \mathcal{W} \) all the comparisons involving two containers in \( p^u \).

Group-based greedy heuristic. The group-based greedy heuristic is a variant of GH, based on the intuition that textual data marked by the same tag tend to have similar contents. To this purpose, \( G^2H \) preliminarily forms groups of containers corresponding to paths ending with the same tag (e.g., ending with name). This corresponds to adjust both the query workload and the similarity matrix to represent inter-group values. GH is launched afterwards on the set of newly formed groups.

Clustering-based heuristic. The clustering-based heuristic employs the classical agglomerative single-link clustering algorithm [Jain et al. 1999]. In our case, the distance between pairs of containers reflects the costs incurred when compressing containers using different algorithms, that in turn depend on the containers’ contents. In particular, the higher are the costs for decompressing a pair of containers, for storing them and their corresponding auxiliary structures, the more distant they can be considered. Moreover, for each algorithm, a decompression cost occurs when the two containers are involved in comparisons not allowed by the algorithm in the compressed domain:

\[
\text{dist}(c_i, c_j) = \sum_{a \in A} \{ \alpha \cdot u_{\mathcal{W}}(a, i, j) \cdot a[cd(F(c_i, c_j))] + \beta \cdot a[cs(F(c_i, c_j))] + (1 - \alpha - \beta) \cdot a[cs(F(c_i, c_j))]} \]

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

CH works as follows. 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 to create partitions having 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 final number of levels bears from a trade-off between execution times and probabilities of finding good configurations (we empirically set this number to 20). Finally, for each generated partition, CH assigns to each set in the partition the algorithm that locally minimizes costs:

1. Let \( \text{dist}_{\text{min}} \) and \( \text{dist}_{\text{max}} \) be the minimum and maximum distance among two containers in \( C \).

2. Divide the range \( [\text{dist}_{\text{min}}, \text{dist}_{\text{max}}] \) into 20 equally-sized sub-ranges.

3. For each sub-range \( r \), if a pair of containers \( c_i, c_j \) exists such that \( \text{dist}(c_i, c_j) \) lies in \( r \),
(a) Build a partition of \( C \) where a pair of containers \( c_i, c_j \) is in the same set if \( \text{dist}(c_i, c_j) \) is less or equal to the minimum value in \( r \).

(b) Assign to each set \( p \) in the partition the compression algorithms \( a \) that minimizes the cost
\[
\alpha \cdot a[c_d(F_p)] + \beta \cdot a[c_s(F_p)] + (1 - \alpha - \beta) \cdot a[0][c_a(F_p)].
\]

(c) Compare the costs of the newly built configurations with the ones compared at previous steps and retain the one with the minimum cost.

Section 6 provides a performance evaluation of the heuristics presented here.

5. QUERY PROCESSING

This section describes XQueC’s query execution engine. In Section 5.1, we describe the physical algebra, query execution model, and physical query operators that we support. Complex result construction is an issue in XQueC, due to its very fragmented storage; we devote Section 5.2 to this issue. Section 5.3 describes XQueC’s query optimization approach, while Section 5.4 compares XQueC with other most closely related XML query processing engines.

5.1 Query execution engine

XQueC’s engine implements a standard \textit{tuple-based} execution model, where operators follow the \textit{iterator} interface [Graefe 1990], consisting of the \textit{init}, \textit{next()}, \textit{get()}, and \textit{close()} methods. Here, \textit{get()} stands for a set of methods like \textit{getInt()}, \textit{getString()}, etc.

We make four extensions to this interface:

1. We add a simple basic \textit{NodeID} data type, consisting of a \{pre, post, level\} triple.
2. We add a simple type \textit{Word}, representing a variable-length string of bits. This type is used for all compressed values.
3. We employ the iterator model that includes some associated \textit{metadata}, describing the properties of the operator’s output, like column number and types, etc. We extend this metadata by attaching to each column whose type is \textit{NodeID}, an integer list \( p\text{List} \) containing the numbers of all paths on which the IDs in that column were found. For example, if the operator column contains IDs of asian item, \( p\text{List} \) will just contain the integer code associated to the path \text{/site/asia/item} . If the operator column returns all item IDs, the \( p\text{List} \) will contain all six path numbers (for items from all continents). For columns whose type is not a \textit{NodeID}, \( p\text{List} \) is empty. Notice that \( p\text{List} \) is attached to an operator, not to an individual tuple.
4. To properly support compression, we furthermore enrich the operator metadata with its corresponding \textit{compression information}, for each column whose type is not \textit{NodeID}. The compression information consists of:
   - The compression algorithm applied to the values in this column, if any. This is a short integer code identifying the compression algorithm used.
   - If the values in this column are compressed, the compression information also contains a pointer to the compression dictionary used for this container, which is also part of XQueC persistent structures.

The compression information attached to the operator’s columns allows to infer which decompression operations, if any, must be performed on values of this field, in order to retrieve the original data.

The physical operators of XQueC can be classified into: data access operators, compression-related operators, and general-purpose operators. We present them in turn.
Data access operators From the storage model described in Section 3, we derive three basic data access operators:

— **IDScan**(path) returns all identifiers of the nodes found on path, in document order. Thus, IDScan interacts directly with a stored ID sequence. Notice that for the needs of query processing, the depth field of the structural IDs must also be filled in from the length of path, although it is not stored in each ID; this is performed by IDScan.

— **ContScan**(valPath) accesses and joins the compressed container corresponding to valPath, and the associated ID sequence, to produce (pre, post, depth, c(val)) tuples, following the order of the pre number.

— **ContAccess**(val, i, θ) iterator, used if an index (e.g., a B+-tree) is established on a container, returns only the (id, value) container pairs such that i θ val, where val is a value and θ is a comparison operator.

Compression and decompression operators To support the integration of compressed simple data types, we include two unary operators:

— **Compress**(op, cols, cis) compresses the values found in columns cols of operator op, based on the compression information provided in cis.

— **Decompress**(op, cols) applies the opposite transformation: it decompresses the data in the columns indicated by the integer array cols of operator op, with the algorithm and compression dictionary designated by the operator’s metadata.

Compress and Decompress are simple pipelined operators, based on the engine’s respective functions; they have also been used in [Chen et al. 2000]. The idea of considering data compression as a physical property, and introducing physical operators to modify it, dates back to Volcano’s generic optimization framework [Graefe and McKenna 1993].

General-purpose XML processing operators XQueC includes regular operators, such as selections, projections, hash– and merge– joins. Furthermore, XQueC includes an n-ary **Merge** operator that, given a number of ordered sequences (typically sequences of identifiers), merges them while preserving their order (and the duplicates that may arise). Finally, XQueC includes grouping and usual aggregation functions (min, sum, first, etc.)

To combine operators based on structural identifiers in their output, XQueC uses several flavors of structural joins, along the lines of [Al-Khalifa et al. 2002], namely:

— The **StackTreeDesc**(op1, op2, i, j) operator described in [Al-Khalifa et al. 2002] combines tuples from op1 and op2 wherever the identifier in op1’s i-th column is an ancestor (or parent) of the identifier in op2’s j-th column.

— **StackTreeDesc-SJ**(op1, op2, i, j) tests tuples for the same condition, but acts as left semijoin, with op1 being considered the left hand side of the join.

— **StackTreeDesc-OJ**(op1, op2, i, j) similarly compares structural IDs, but acts as a left outerjoin, with op1 being considered the left-hand operand of the join.

StackTreeDesc, StackTreeDesc-SJ and StackTreeDesc-OJ are needed in XQueC to implement XQuery semantics. We have implemented the StackTreeDesc-SJ and StackTreeAnc-OJ in XQueC in the aftermath of our prototype demonstration [Arion et al. 2003]; the need for such operators was advocated in [Chen et al. 2003].

Fig. 3 exemplifies XQueC’s basic and more complex operators, and their metadata. In the XML snippet at the top left, each element appears preceded by its structural IDs. For the time being, consider only the operators aligned in the lower half of the figure; we will discuss the SortedOuterUnion operator shortly. Next to each operator in the lower half, we depict its output tuples on the given XML snippet. The metadata information associated
Fig. 3. Sample physical operators and corresponding operator metadata in XQueC.

Table III. Query execution plans corresponding to Join1 - Join4 in Fig. 3.

to each operator column is shown in a rounded-corners dashed box, and is connected by a dashed arrow to the respective column. For instance, the leftmost IDScan operator returns the two identifiers corresponding to person elements; the type and the path of such identifiers are shown in the associated rounded-corners box. The next operator is a ContScan, returning pairs of identifiers and compressed values, where C(s) stands for the compressed value of string s.

The plans denoted as Join1 and Join3 pair identifiers of person elements with the identifier of their name children. Finally, the plans denoted as Join2 and Join4 in Fig. 3 combine the identifiers of person with those of their name (resp phone) children, and with the textual values inside the children. Possible query plans producing the output and the metadata associated to the plans Join1 to Join4 are depicted in Table III.

5.2 Constructing XML results from XQueC’s storage

XQueC’s storage model exhibits a strong degree of vertical fragmentation. This poses particular challenges to its execution engine, in two situations: when decompressing (fragments of) the original document, and when building arbitrary query results, including new elements. In both these situations, complex, nested data must be built, out of several fragments, issued directly from the store, or produced by more complex query plans. Notice
that the former is not an issue in systems storing a persistent tree [Jagadish et al. 2002; Fiebig et al. 2002; Halverson et al. 2003], nor for homomorphic compressors such as XGrind [Tolani and Haritsa 2002]. However, the latter is an issue for all systems supporting new result construction. We describe here XQueC’s approach for the task of combining such fragments (each coming from an iterator) into a potentially complex XML structure.

XQueC’s execution engine provides two different operators for this task. The first one is an adaptation of an existing technique for exporting relational data in XML; the second one specifically leverages XQueC’s fragmentation strategy and structural identifiers. Interesting trade-offs exist between them, as we explain next.

The SortedOuterUnion approach

This approach is inspired from [Shanmugasundaram et al. 2000], but it is specialized to the task of assembling flat, narrow tuples obtained from the XQueC storage, into a hierarchical structured format.

XQueC’s SortedOuterUnion operator takes in input $n$ operators whose tuples have varying length and are organized in a well-defined manner, and constructs a single output by labeling the input tuples and padding them conveniently with — (null) fields. An example appears at the upper half of Fig. 3. This SortedOuterUnion assembles the content from its six child operators in a single set of output tuples (shown at the top of the figure), having exactly the hierarchical form of the XML snippet at their left.

The SortedOuterUnion children operators must provide the information needed to construct hierarchical XML data. More specifically, there must be a child operator for each element type to be constructed, and each such operator must be followed by others providing all the values (text leaves or attribute nodes) to be written into such elements. For instance, in Fig. 3, the children IDScan, Join1 and Join3 provide such ID tuples; furthermore, ContScan follows IDScan, Join2 follows Join1, and Join4 follows Join3.

The SortedOuterUnion labels tuples of each child by appending an integer field equal to the child’s position, null-pads tuples from each child, up to the signature obtained from the natural joins of all child signatures, and sort-merges all the tuples, in the lexicographic order dictated by both their ID fields, and the numerical label. This is performed in a single pipelined step. Data values manipulated by the SortedOuterUnion may still be compressed: decompression may take place later.

A pipelined XMLize operator added on top of the SortedOuterUnion to construct new elements, based on a tagging template describing the result structure.

The SortedOuterUnion-XMLize approach has the disadvantage of requiring intermediary “join products”, such as the plans Join1 to Join4 in Fig. 3 (see also Table III). Furthermore such intermediary join plans may share subtrees; for instance, IDScan(person) is an input to all plans from Table III. Multiple outputs go beyond the classical iterator model [Graefe 1990], thus, we execute shared subtrees once, materialize the result, and subsequent consumers read it multiple times. XQueC’s reconstruction plans based on SortedOuterUnion are built so that only ID-only plans are materialized; values, in contrast, are always joined last, and never stored as intermediary results. However, such execution still has significant memory needs, and furthermore, it breaks the pipeline due to materialization. These problems are overcome by the second reconstruction method, which we describe next.

Direct reconstruct

This approach relies on an $n$-ary Reconstruct operator, receiving as inputs a number of decompress(ContScan) and IDScan operators, such that the path of one IDScan is an ancestor of all other paths associated to the input operators. We will call this path root path. The effect of the Reconstruct is to combine the decompressed values,
and at the same time tag them in a way that respects the paths of the original document (we recall that these paths are available as metadata associated to the operators).

When a Reconstruct operator is created, it inspects its children and organizes them in a hierarchical map, mirroring the operators’ associated paths. The map contains one node per IDScan input. Each such node is associated: (i) the respective IDScan, say for path p; (ii) possibly a set of decompress(ContScan) operators, corresponding to the attribute paths of the form p/@attr; (iii) possibly a decompress(ContScan) corresponding to p/#text; and (iv) possibly an ordered list of children map nodes, for all paths p’ represented by an IDScan in the input, such that p’ is a descendant of p, and no other path between p and p’ is represented in the input. A map node also contains a 1-size buffer for each of its associated operators, used for value lookahead. The order among sibling map nodes is dynamic: at any point during the execution, it reflects the order among the pre numbers found in their (pre, post) buffers.

For example, the Reconstruct operator is illustrated in Fig. 4 output the XML snippet in Fig. 3. The operator’s map has three nodes, shown as rounded boxes. The dashed arrows follow the flow of data: each map node makes next() calls to its associated operators and buffers the result in the 1-size buffer. The Reconstruct itself reads values directly from all the buffers associated to the inputs, and orders them correctly in the output, guided by the path metadata and pre and post numbers.

The Reconstruct operator runs in pipeline. It uses memory for the 1-value buffers, and for the map structure, thus linear in \(|\text{in}|\), the number of input operators. The CPU cost per output tuple is of the order of \(O(\log_2(|\text{in}|))\).

**Trade-offs** The advantages of the Reconstruct are its non-blocking nature, and its low memory footprint; this makes it preferable to the SortedOuterUnion-XMLize approach in the case of element reconstruction. However, the Reconstruct is driven directly by the order of IDs in the input; thus, it cannot be used to produce an output in an order that disagrees with the input order. This would be the case, for instance, if we wished to output the person elements in Fig. 3 with their phone children before their name children. This problem is not encountered by the SortedOuterUnion-XMLize approach, which relies on multi-column join products and explicit integer labels to order the components of the output. These trade-offs are taken into account by the XQueC optimizer, as the next section shows.

### 5.3 Query optimization

In this section, we outline the process of XQuery optimization in XQueC.

Optimization proceeds in three stages: binding, combining, and output construction. These three stages are dictated by the general structure of XQuery queries: query variables, and variables appearing in path expressions, must be bound, i.e., the corresponding data values and/or element IDs must be identified in the storage. Then, the bindings must be combined, according to e.g., the structural relationships between the node IDs designated by the variables, or the value-based predicates specified by the query. Finally, the output
must be constructed. We consider each stage in turn.

**Variable binding**

The general process of binding variables to the correct (and minimal) set of useful paths, is based on the structure summary information described in Section 3. To formalize this process, we first introduce the useful notion of *query patterns*:

**Definition 5.1.** Let \( \xi \) an XPath expression, built over the fragment \( \{ /, /, +, [] \} \). The query pattern of \( \mathcal{P}(\xi) \) is defined as a tree pattern, with a unique root node, labeled \( \top \) if the expression starts at the document root, or with a variable name, if the expression starts from a variable; edges whose parent or child node are conditional nodes (appear inside \( [ ] \) in \( \xi \)) are furthermore labeled \( \circ \), standing for semi-join; the other edges are labeled with \( j \), standing for join. As in tree patterns, double (single) edges represent descendant (child) axes.

For instance, a simple XPath expression and its pattern are depicted in Figure 5(a).

We consider XQuery expressions characterized by the restricted grammar of [Paparizos et al. 2004]. Basically, such expressions feature XPath patterns as just defined above, and FLWR expressions arbitrarily nested. XPath expressions appearing in an XQuery may start either from the document root, or from a query variable. We say an XPath expression \( \xi_1 \) depends on another expression \( \xi_2 \) if the former is used to bind a variable, while the latter starts from that variable.

Let \( \xi \) be a restricted XQuery expression as described in [Paparizos et al. 2004]. The query pattern of \( \xi \), denoted as \( \mathcal{P}(\xi) \), is a forest of XPath tree patterns, and a set of joins (or structural joins) connecting these patterns. For instance, Figure 5(b) shows a sample XQuery expression; its elementary XPath expressions and their corresponding patterns are in Fig. 5(c), while the pattern for the XQuery expression is in Fig. 5(d).

**Definition 5.2.** Let \( \xi \) be an XQuery query as above, and \( PS \) be a path summary. A match for \( \mathcal{P}(\xi) \) over \( PS \) is a function \( \phi \), associating to each node \( n \in \mathcal{P}(\xi) \) a set of \( PS \) nodes \( \phi(n) \) as follows. For every \( n \), and every \( p \in PS \), \( p \in \phi(n) \) if and only if, for some document \( D' \) such that \( PS(D') = PS \), elements on the path \( p \) in \( D' \) may contribute to the result of \( \xi \) over \( D' \).

A match for \( \mathcal{P}(\xi) \) associates to every pattern node all possible paths in the path summary to which this node could be bound, *given the path constraints encapsulated in the path summary*. For instance, Fig. 5(d) shows a match of the pattern extracted from the XQuery in Fig. 5(b), against the path summary from Fig. 1(b). For every node \( n \in \mathcal{P}(\xi) \),
the paths in $\phi(n)$ are shown in a rounded dotted box, connected by a dotted line to $n$.

Computing a match of a query pattern against a path summary amounts to evaluating the query against the summary, similar to Lore [McHugh and Widom 1999], but with patterns whose edge have various join, outerjoin, and semijoin semantics.

We have implemented two methods for computing the match. The first one is a na"ive in-memory evaluation, based on a top-down traversal of the path summary, suitable for small summaries. The second one is a streaming evaluation of the query pattern, viewed as a forest of individual tree patterns. We have implemented our own stack-based lazy deterministic automaton in the spirit of [Green et al. 2003], matching all nodes of a forest of tree-shaped patterns. The automaton-based evaluation computes a query pattern match in $O(N_{PS})$ time, using at most $O(|P(xq)| * N_{PS})$ space. It has been shown in [Green et al. 2003] that the actual space needs are lower for many real-life settings. This second method is suitable for the cases where the summary is too large to be handled in memory.

Given a query such as the one in Fig. 5(b), the optimizer computes the match of its pattern against the path summary. Then, for every variable and path expression, in our case $\$i$, $\$d$, $\$i//keyword $\$i/name$, and $\$d//emph$, it constructs binding plans as follows. Let $n$ be the query pattern node for such a variable or expression. If $\phi(n)$ contains just one path, the optimizer builds an IDScan operator. If $\phi(n)$ contains several paths, the optimizer builds a Merge operator over IDScans. Finally, if there is a value predicate associated, the optimizer constructs select(ContScan) or ContAcc plans, instead of IDScan.

In the presence of a value selection, such as the one on $\$i//keyword path expression, the XQueC optimizer constructs a plan as follows:

---

- The optimizer identifies the compression method applied on the container /site/regions/asia/item/description/keyword/#TEXT, and its compression dictionary.
- The compression of the string “romantic” is triggered, using this compression algorithm and dictionary; let $w$ be the compressed result.

A plan of the form ContAccess(/site/regions/asia/item/description/keyword/#TEXT, $w$, $=$) is constructed as binding for the $\$i//keyword path expression.

Combining The optimizer combines all binding plans through structural joins, semi-joins and outerjoins. The bindings for the variables and expressions in the for-where clauses must participate to several outerjoins, corresponding to several expressions in the return clause. To avoid joining emph bindings with name ones, and introducing multi-valued dependencies, the optimizer inserts Read and Materialize operators at this point; the resulting plan is shown in Fig. 6 at left. Filled round bullets represent Materialize operators, while empty bullets designate Read operators; operators in a producer-consumer relationship are connected by dashed lines. Notice the graph shape of the plan.

This initial plan is derived from query syntax. Subsequently, the optimizer applies equivalent join reordering transformations. XQueC’s simple cost model is based on cardinality estimation, derived from the statistics defined in Section 3.2, as in [Aboulnaga et al. 2001].

The join reordering results in the plan shown in Fig. 6 in the center. Notice that in this plan, the bottom leftmost operator is the most selective one, namely the ContAccess.

Using path metadata to avoid sorts and duplicate eliminations As previously explained, the StackTreeDesc and StackTreeAnc structural join operators produce their outputs in the descendant, respectively the ancestor ID order. To avoid sorting steps, it would be interesting to have a structural join preserving both ancestor and descendant ID order; however, this is not always possible. A structural join preserves both input orders iff the
IDs in the ancestor are free of ancestor-descendant pairs [Al-Khalifa et al. 2002; Chen et al. 2003]. This condition is data-dependent, and thus impractical to check. Thanks to the path metadata associated by XQueC to each ID column, a simpler condition can be verified:

Let \( \text{op}_1 \) and \( \text{op}_2 \) be two operators, such that \( \text{op}_1[i] \) and \( \text{op}_2[j] \) contain element IDs. If all paths associated to \( \text{op}_1[i] \) are pairwise unrelated, then a structural join matching ancestor IDs in \( \text{op}_1[i] \) with descendant IDs in \( \text{op}_2[j] \) will preserve both input orders.

XQueC’s optimizer checks this simple condition before deciding whether the output of a structural join may need reordering.

Handling predicates in the compressed domain We now explain how XQueC handles joins and other predicates (such as string inequality, etc.) on compressed values.

Consider the join \( \Join (\text{op}_1; \text{op}_2; \text{rel}) \). If the join columns in \( \text{op}_1 \) and \( \text{op}_2 \) are compressed with the same method, and furthermore the \( \text{rel} \) predicate can be evaluated on compressed values, then XQueC simply constructs the join. Otherwise, XQueC decompresses both join columns prior to the join, by inserting Decompress operators only affecting those columns. The reasoning for selections is similar.

XQueC applies lazy decompression: we only decompress data as late as possible in the flow of execution. An alternative approach is transient decompression [Chen et al. 2000]: decompress some compressed item (e.g., to apply a comparison unsupported by the compression method), apply the comparison, then compress again the values, in order to manipulate smaller intermediary results for the remainder of query processing. Transient decompression could be incorporated into XQueC’s optimizer, by adding the corresponding plan transformation rules (introducing the Decompress and Compress operators as explained above). We stress, however, that XQueC chooses compression algorithms allowing the evaluation of predicates in the compressed domain. Thus, one of the motivations for using transient decompression is diminished; the other motivation (handling smaller intermediary results) is of course valid. We demonstrate some of the trade-offs involved between lazy, eager, and transient decompression in Section 6.4.3.

Output construction The optimizer finalizes the plan based on a mixture of a SortedOuterUnion-XMLize operator to construct the new structure required by the query, and Reconstruct to copy input subtrees to the output, e.g., $i/name$ and $d//emph$ in Fig. 5(b). The larger these subtrees, the more interesting it is to use Reconstruct operators. Fig. 6 depicts a complete plan, based on SortedOuterUnion only.
5.4 Query processing in XQueC vs. other XML query engines

In this section, we compare XQueC’s query engine to existing similar engines.

XQueC is a system endowed with a persistent store, which discourages comparison with in-memory systems such as described in [Buneman et al. 2003] or [Marian and Simeon 2003]. Also, XQueC’s query engine follows a classical \textit{iterator} model, provides \textit{selective data access}, and puts an emphasis on having \textit{low memory requirements for query processing}. We view these features as strictly necessary if the system is to handle large documents, and if the machine’s memory is to be shared between several applications and/or users. Finally, XQueC relies on \textit{pre, post structural identifiers and structural joins}.

For all these reasons, we consider XQueC directly comparable with systems such as Natix [Fiebig et al. 2002], Timber [Al-Khalifa et al. 2002; Jagadish et al. 2002] and Niagara [Halverson et al. 2003]. These systems store XML documents using two persistent structures. The first one is a \textit{persistent tree image} of the document, organized in nodes connected via disk-based pointers. The second is a \textit{tag-partitioned B+-tree index of structural identifiers}, which gives access to all \textit{pre, post} IDs of a given tag. We call this second index structure a \textit{tag-partitioned index}, or more simply \textit{tag partitioning} (TP).

In contrast, XQueC does not store a persistent tree, which reduces its disk footprint, at the expense of extra effort for document reconstruction. Also, ID sequences are more compact than B+-trees, since the latter use intermediary pages and leave space to accommodate future insertions. This allows XQueC to actually compress an XML file, as opposed to the increased storage size reported in [Halverson et al. 2003] and discussed in Section 3.

As previously explained, XQueC applies \textit{path partitioning} on its \textit{pre, post} identifiers, and uses this structure not as an index, but as the actual store. For the purpose of this section, we call this approach \textit{path partitioning} or PP. We outline the differences between XQueC and systems such as Timber and Natix at various stages of query execution: binding, combining, and reconstructing.

\textit{Binding variables with TP and PP} Consider binding a variable to a simple path expression of the form \texttt{//@l1//@l2//...//@lk}.

XQueC will compute a pattern from this expression, then find a pattern match against the path summary, and then, based on its PP storage, construct a \textit{Merge} plan over a set of \texttt{IDScan} operators. The identifiers of the respective elements will be returned in document order, and without duplicates.

With TP, the approach described in [Wu et al. 2003; Halverson et al. 2003] requires: \textit{(i)} Accessing all IDs of elements labeled \texttt{l1}, \texttt{l1}, ... \texttt{lk}. Notice that this access is not very selective, since many elements labeled \texttt{l} may not participate to the path. \textit{(ii)} Combining these elements via \texttt{k} – \texttt{1} structural joins, or alternatively a single holistic twig join [Bruno et al. 2002]. \textit{(iii)} Potentially, a duplicate-elimination step and/or a sorting step may be necessary to ensure the identifiers are in the correct order and duplicate-free after the joins.

The unselective access, bigger join effort, and potential duplicate elimination and sorting steps are handicaps of the TP approach, and make variable binding much more efficient on a PP store. The PP approach, on the other hand, must merge various ID sequences, which is not needed with TP. Overall, variable binding is in some cases much more efficient using PP. We demonstrate this experimentally in the Appendix (Section 8.1).

\textit{Combining binding plans with TP and PP} XQueC implements some of the structural join algorithms proposed originally in Timber [Al-Khalifa et al. 2002]. \textit{HolisticTwigJoins} [Bruno et al. 2002] could be easily added. Binding combination in XQueC
differs from similar systems by its usage of path metadata for two purposes.

First, XQueC uses path metadata to avoid, in some cases, sort and duplicate elimination steps over the output of a structural join, as explained in Section 5.3.

Second, XQueC may use skipping structural joins (also known as zig-zag [Halverson et al. 2003]), to avoid reading parts of an input which have no match in the other input [Manolescu et al. 2004]. Typically, skipping was achieved based on B+-tree indexes on the pre ID field. In XQueC, compression rules out B+-trees, thus skipping is supported directly by binary search over the ordered sequences. Furthermore, XQueC uses path metadata to decide when it is safe to use skipping also on post [Manolescu et al. 2004].

From the viewpoint of skipping structural join inputs, XQueC also compares with the XPatch accelerator technique [Grust 2002], which uses numerical properties of pre and post numbers to avoid useless parts of a join input. XQueC’s path-partitioned storage reduces the need for such accelerators, since the IDs that XQueC will attempt to join are already guaranteed to be on corresponding paths; this greatly reduces both input sizes and join effort, as we demonstrate in the Appendix. In any case, the index space requirements of an XPath accelerator are incompatible with XQueC’s compression goal.

**Result Reconstruction** Due to its lack of a persistent tree, XQueC is handicapped in this respect, when compared with [Jagadish et al. 2002; Fiebig et al. 2002; Halverson et al. 2003]. These systems only need to copy subtrees from the input to the output, while XQueC must make a reconstruction effort.

We have presented XQueC’s dual approach for reconstructing the document, and for building new results, in Section 5.2. We expect a persistent-tree storage to be more efficient when extracting data from the store. The remaining effort needed by all systems, regardless of whether they use a persistent tree or not, is due to new element construction. In this respect, XQueC adapts a relatively standard technique [Shanmugasundaram et al. 2000].

XQueC’s Reconstruct operator can be compared to XMill’s decompressor [Liefke and Suciu 2000], which exploits a structure stream. In XQueC, there are many stream, corresponding to individual paths, thus, the Reconstruct uses the hierarchical map to assemble the input streams. Also, Reconstruct can read only parts of the data, unlike XMill.

The Reconstruct is also comparable with input tree reconstruction from a node-oriented store [Grust 2002]. In that store, all element IDs are clustered in document order, thus no stream merging is needed. However, that approach is only useful for copying input trees as such, not if one wishes to extract, e.g., person elements without their phone children. In this case, the approach of [Grust 2002] would need to skip inputs, and to use a quite large ID index, which is not usable in XQueC.

### 6. EXPERIMENTAL ASSESSMENT

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

We have implemented this storage and compression strategy within our XQueC system prototype. Section 6.1 sets the background for our experimental study: we introduce the architecture of our prototype, the data sets and computers we used for our experiments, and the systems we compare XQueC with.

We then 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. The results are reported in Section 6.2.

—Compression factors: we have performed experiments on both synthetic data (XMark
We conclude in Section 6.5.

files. We implemented both options. Set of documents we consider for our measures, and introduces a set of useful notations.

Again, the compressed containers may be backed by in-memory vectors, or disk-resident — Query execution times: we show how our system performs on some XML benchmark — System implemented in Java, and it currently accounts for approx. 50,000 lines of code. The system

In this section, we describe the architecture of the XQueC prototype, which we demonstrated in [Arion et al. 2003]. XQueC is implemented in Java, and it currently accounts for approx. 50,000 lines of code. The system modules are outlined in Fig. 7 (left).

The XQueC loader. The loader decomposes the XML document into ID sequences and containers, as described in Section 3. The sequences and containers must be accessed in append mode. We have implemented two versions of these structures: one using in-memory vectors, one based on disk-resident structures. The disk-based structures can always be used, and in particular for very large documents, because the resulting memory consumption is low. However, intensive disk I/O tends to slow down the loading. For small-to-moderate size documents, faster in-memory structures can be used.

The XQueC compressor. The compressor partitions the data containers according to the strategy described in Section 4, then applies the resulting compression on the containers. This phase produces a set of compressed containers, and a set of compression dictionaries. Again, the compressed containers may be backed by in-memory vectors, or disk-resident files. We implemented both options.

The ID sequences, path summary, compressed containers and compression dictionaries are all loaded in the XQueC persistent repository.

Fig. 7. Architecture of the XQueC prototype (left), and sample query plan image produced by XQueC (right).
The compressed repository. The repository stores the compressed documents and provides: (i) compressed data access methods, and (ii) a set of compression functions that use persistently-stored compression dictionary to compress at runtime constant values appearing in the query. The usage of such functions was illustrated in Section 5.3.

We used as back-end an embedded database, Berkeley DB [BER 2003], providing a set of low-level persistent storage structures, in which items can be stored, associated to keys. To store ID sequences and containers, we used Berkeley DB’s persistent sequences: fixed length for ID sequences, and variable length for containers. We store the pre, post values of an ID compacted together in the minimum number of bytes necessary. Containers and ID sequences are clustered into data volumes, each of which is set to contain at least 50,000 entries (individual collections in a volume are directly accessible).

Inserting data in a fragmented, persistent store is significantly slower than appending to a single file, as some similar systems do. A better buffering mechanism for compressed container is also needed, and currently under development.

The query processor. The processor includes the physical operators listed in Section 5.1 and Section 5.2, and the query optimizer outlined in Section 5.3. The XQuery subset that our prototype supports does not include features like typing, validation, support for XQuery functions etc. A QEP produced by XQueC for a simple query is represented in Fig. 7 at right, topped by a SortedOuterUnion-XMLize pair.

6.1.2 Data sets and Machines. In the sequel, we will use several XML data sets, whose name, size and provenance are listed in Table IV. The INEX document consists of a corpus of IEEE computer science publications, formatted in XML; this document is used as a test base within an XML information retrieval benchmark. The documents named XMark_n are generated using the XMark generator, at the desired size in MB. For comparing with XPRESS, we also include the documents they used [Min et al. 2003] ShakespeareXPress is the Shakespeare dataset multiplied 2 times; 1998statXPress is the 1998fullbaseball dataset, multiplied 16 times and, finally, WashingtonXPress is the UW course dataset, multiplied 4 times. The experiments reported here were undertaken on three distinct computers, with the following characteristics. Machine M1 has a 1.7 GHz processor, 512 MB RAM, and is running Windows XP. Machine M2 has a 1.4 GHz processor, 1 GB RAM, and runs RedHat 9.0. Finally, M3 has a 1.13 MHz processor, 2GB RAM, and runs RedHat 7.2. For simplicity, we will refer to these machines in the sequel by their name, e.g., M1.

6.1.3 Competitor systems. As a general observation, many existing XML queryable compressors are either covered by copyright (e.g., XPRESS [Min et al. 2003], XQZip [Cheng and Ng 2004]) or unusable for large datasets (e.g., XGrind [Tolani and Haritsa 2002]). Thus, a comprehensive comparison was not easy. We briefly outline, for each competitor systems, the comparisons that we were able to perform.

XMill is available for download. It is a non-queryable compressor, thus the comparisons we could perform were: compression factors, compression time and decompression time. XGrind is also available at http://sourceforge.net). We compare our compression and decompression performances, on the documents shown in Table IV. Unfortunately, we could not run any query on these documents, since the query processor crashed, including on the documents and queries included in the download package.

The XPRESS compressor is not publicly available. Thus, we use the compression factors from [Min et al. 2003] in the comparison. We did have access to an earlier version of the
code, on which, however, all whitespaces were lost by compression and decompression. This should be taken into account when comparing the compression factors.

Similarly, XZZip is not available and we used the compression factors from [Cheng and Ng 2004]. We note, however, that the XMark data sets used in [Cheng and Ng 2004] have been “simplified” by eliminating the structure of rich textual types, such as item descriptions. On XMark 111, this eliminates 5 out of 12 levels of nesting, and reduces XMark to little more than a relational data set. This should be kept in mind when comparing the CFs.

The queriable compressor described in [Buneman et al. 2003] is available. It does not directly compare with XQueC since it does not produce a compressed persistent structure, and it does not compress values. We compare the size of its compressed structure with our path summary.

### 6.2 Compression Choices

**Characterization of compression algorithms.** As highlighted in Section 4, each compression algorithm is characterized by a cost vector \(< c_d(F), c_s(F), c_a(F, \sigma) >\), whose components indicate the costs of decompression \((c_d(F))\), compressed container occupancy \((c_s(F))\) and auxiliary structures occupancy \((c_a(F, \sigma))\). These costs being the cost model input, are a function of a container similarity matrix \(F\); in Section 4.2.1 we also explained how \(F\) has been assessed for our experiments. The costs of the compression algorithms considered have been measured on synthetic containers filled with strings of up to 20 characters each; the total containers 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.

**Evaluation of the studied heuristics.** The heuristics presented in Section 4.2 aimed at making XQueC capable of building, with little computational effort, compression configurations improving over the naïve ones. In this section, we evaluate the \(GH\), \(G^2H\), and \(CH\) heuristics, by comparing the compression configurations they produce against the
naïve ones described in Table V. We used three sample workloads, shown in Table V, and the containers as extracted from XMark75. The first workload is a subset of the XMark benchmark workload; RW1 and RW2 were randomly generated based on the containers extracted from the same document. We also analyzed a no-workload case to show the quality of compression results in the absence of a workload. In the experiments, performed on M1, we considered two possible assignments for the cost function weights: $\alpha = 1, \beta = 0$, for the case when only the decompression costs are considered; $\alpha = 0, \beta = 0.5$, where the container and source model storage costs are considered, and equally weighted.

![Fig. 8](image)

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

We report the results in Figure 8. They show that in the majority of cases, the cost of one of the configurations obtained by running the heuristics is kept lower than the costs of the naïve configurations. Moreover, all the proposed heuristics happen to run very fast, as expected; the total execution time was 158.04 seconds at most for the cases with $\alpha = 1, \beta = 0$, 54.98 seconds for the cases with $\alpha = 0, \beta = 0.5$, and 8.08 seconds for the no-workload case. Therefore, a suitable solution could be that of running them together, then grab the best one among the obtained configurations.

### 6.3 Compression and decompression performance

In this section, we report the performances of XQueC with respect to XML data compression and decompression. Section 6.3.1 discusses the sizes of various compressed data...
structures produced by XQueC, and demonstrates the impact of the compression configuration on these sizes. Section 6.3.2 presents measures of XQueC’s compression time. We compare XQueC’s compression factor with respect to similar systems in Section 6.3.3. We present XQueC performance on data decompression in Section 6.3.4, and close this section by comparing XQueC’s compression and decompression times with similar systems, in Section 6.3.5.

6.3.1 Compression factors. Fig. 9 shows the sizes of the compressed data structures produced by XQueC, and the compression factor attained, for the documents from Table IV, using the NaïveHuffman1 compression configuration. For readability, we have plotted separate graphs for relatively small documents (up to 25 MB) and for the larger ones (up to 483 MB). We make the following remarks.

In Figure 9, documents mainly contain structure (tags), and string values; numerical values have insignificant sizes. This characteristic is thus confirmed over a varied dataset, which justifies focusing our compression choice efforts on strings.

The relative importance of containers and structures varies with the nature of the document (see the document description in Table IV). Our encoding of structure into ID sequences and a path summary reduces the structure size by a factor varying between 2 and 4, with the exception of the TreeBank data set, for which we achieve a factor of 1.23. TreeBank is also an outlier, due its large path summary (6 MB), the only one large enough to be visible in Figure 9.

The (string) container compression factor achieved with NaïveHuffman1 varies depending on the data set. The overall compression factors from Figure 9 are only given as a baseline for the overall compression efficiency; we will discuss them in more details next.

Among all the compressed data structures that XQueC builds, the most important are always the string compressed containers, given the weight of string data types inside typical XML documents. XQueC invests a special effort in choosing the compression configuration for string containers, as described in Section 4. The next set of measures examines the impact of compression configurations on the resulting compressed data structure sizes.

Varying compression configurations. Figure 10 shows the total size of compressed containers, and of compression dictionaries, for nine compression configurations previously introduced. We have isolated smaller documents (top and bottom left) from larger documents (top and bottom right), for readability. The configurations used here are compression-oriented (do not depend of a workload). For SwissProt and DBLP, we did not measure the effect of NaïveALM2, for reasons explained shortly.

Fig. 9. Sizes of compressed data structures produced by XQueC and compression factor (NaïveHuffman1 compression configuration).
Figure 10 shows that the compression configuration does indeed impact the resulting compressed structure sizes. Let us discuss some of the trade-offs involved. Among the naïve configurations, those based on ALM, and in particular NaïveALM2, tend to achieve the strongest container compression. The reason is that ALM exploits repetitive substrings for compression. However, considering the dictionary size for the same configurations tells a different story. NaïveHuffman1 wins unsurprisingly: a single dictionary for all characters is clearly small. NaïveHuffman3 and NaïveHuffman2 start losing this advantage, especially for complex documents such as XMark111 and DBLP, since the dictionary size is multiplied by the number of containers (shown in Table IV). The most striking difference, comes from NaïveALM2, which produces very large dictionaries, by extracting frequent strings from each container. The dictionary size reaches 1.8 MB for the Nasa document, which is important when compared with a compressed data size of 6.5 MB; not only it loses the space saved by NaïveALM2 over, say, NaïveHuffman1, but furthermore, this space is now needed in memory (for NaïveALM2 dictionaries) instead of disk. NaïveALM2 produces very large dictionaries: for XMark111 they total 11.7 MB, and for SwissProt and DBLP, the compressor’s memory requirements were so high that loading failed. This highlights the importance of considering more than just the size of the compressed strings, when choosing a compression configuration.

The three heuristics, $CH$, $GH$ and $G^2H$, make smart compromises between the conflicting objectives of small containers (intuitively, better served by small partition groups, and ALM), and of small dictionaries (better served by large partition groups, and Huffman). This highlights the need for a cost-based configuration search, which is the only way to balance such objectives. The compressed container sizes are overall better than simple Huffman-based ones; the dictionary sizes are consistently smaller than the bad cases, especially in the case of $G^2H$, grouping values in same-tag containers.

From these measures, we draw the following conclusions. A good choice of a compression configuration strongly impacts both the compressed container sizes, and the dictionary sizes. Trade-offs are already apparent with only two compression algorithms; adding more
algorithms would further complicate the choice. XQueC was built to handle XML data even in the absence of value type information. If such information was available, presumably many of the values we treated as strings would be associated specialized types, which may benefit from a variety of specialized compressors, as originally done in XMill [Liefke and Suciu 2000]. Thus, a proper cost-driven choice of the compression configuration will be of even more importance in such settings.

6.3.2 Compression time. XQueC functions as a full-fledged DBMS, endowed with a persistent store and with a query processor. As explained in Section 6.1.1 (see also Fig. 7), loading a document within XQueC’s store consists of three stages:

— extracting a set of containers, ID sequences, and path summary from the document;
— compressing the containers;
— loading the compressed containers, compression dictionaries, ID sequences, and path summary in the BerkeleyDB storage.

We start by studying XQueC’s performance for compression and decompression, following the above cost decomposition, for increasingly large documents.

For this measure, we used XMark documents, and the NaïveHuffman1 and NaïveALM1 compression configurations from Table V, since they represent extreme cases: Huffman compression provides fast compression and small dictionaries, while ALM is slower in compression and faster in decompression.

To distinguish costs associated to document partitioning and storage, from compression costs, we also used a Dummy compression configuration in which the “dummy” compressor copies Strings into words, byte by byte, without applying any compression. The measures were made on M3, and we set the BerkeleyDB page cache size at 4 Kb. We used in-memory containers and ID sequences. The results are reported in Fig. 11, which shows:

— the parsing time, including the time to parse the document, fill in the containers and ID sequences, and the path summary;
— the container compression time. This does not include the time to pick the compression configuration, since in this measure, the compression configuration is hardwired;
— the time to store the compressed containers in the repository;
— the time to store the document structure in the repository;
— the time to store the compression dictionary and path summary in the repository (this is typically small); and
— the total loading and compression time.

We notice that the various time components grow linearly with the document size.
The compression time amounts to about 70% of the total time with naïve ALM compression, 25% with naïve Huffman compression, and less than 10% in the case of Dummy compression. The compression time of the Dummy compressor reflects the time spent converting the data from string to byte arrays acceptable to BerkeleyDB (necessary to escape possible null characters in the input, which are interpreted as end-of-stream by BerkeleyDB). The time difference between the Dummy and Huffman or ALM compressors, corresponds thus to the actual compression effort. The compression time is more important with ALM, which makes more effort in identifying the frequent strings in the input. In contrast, a Huffman dictionary is simply constructed by assigning one code to each character. XQueC is meant for the scenario where data is compressed and loaded once, and is subsequently queried many times; thus, we consider the ALM compression time a reasonable price to pay for the algorithm’s good properties, notably order preservation and fast decompression.

Third, the time to load the data is not negligible, since we use a persistent store based on BerkeleyDB. Queryable compressors using an in-memory document image, or based on a SAX parser, trade query capabilities and scalability against this overhead.

Fourth, the time to load the path summary (and dictionaries) is negligible. This is due, first, to the small size of the summary, and second, to the modest size of compression dictionaries, when compared to the data size.

6.3.3 CF comparison with competitor systems. We now compare XQueC’s compression factor (CF) with that of competitor systems. We performed two sets of measures.

First, we compare XQueC’s CF with: those of XQZip from [Cheng and Ng 2004], and those that we measured ourselves using XMill and XGrind. For space reasons, we only show results for the NaïveHuffman1, NaïveHuffman2 and NaïveALM1 configurations.

Figure 12 (top) shows that XQueC CF is slightly inferior to that of XQZip and XGrind; the overall comparison depends on the data set. We consider XQueC’s CF to be acceptable given the query capabilities that the system offers. Indeed, it is closer to an XML database, than to an in-memory compression tool such as XMill (which does not support querying), and XGrind and XQZip (which only support limited XPath queries). XQZip’s compression ratio is in part due to its use of gzip over individual data blocks. Interestingly, the values reported in [Cheng and Ng 2004] as being XGrind’s CF are lower than the ones we measured ourselves, and which we report in Figure 12 (top).

Second, we compare XQueC with XPRESS, XMill, and XGrind; since for this comparison we had to rely on the compression factors reported in [Min et al. 2003], we had to use those data sets for XQueC, XMill and XGrind. Figure 12 (bottom) shows that XQueC CFs are rather comparable to those of XPRESS and slightly worse than XGrind. Keep in mind that these data sets have all been obtained, as in [Min et al. 2003], by multiplying original data sources several times; we do not believe such sources are very representative.

6.3.4 Decompression time. Data decompression in XQueC amounts to executing one of two alternative query execution plans, Sorted Outer Union and Reconstruct, described
Fig. 13. Reconstruction query times.

in Section 5.2. Thus, decompression time can be split in: data read time; value decompression; and the remaining processing effort which depends on the query plan.

We start by examining the impact of various compression configurations on the execution time of decompression queries. Figure 13 illustrates this on varying-size XMark documents, for the queries \( QR_1 \), \( QR_2 \) and \( QR_3 \) (Table IV). The Reconstruct approach was used to produce the results; measures were performed on machine \( M_2 \). We compare the heuristics \( CH \), \( GH \) and \( G^2H \) with the Na"iveHuffman baseline, which can be considered as an XQueC “worst case”, since its compression factor is not very good, and decompression is slower than for ALM.

The graphs in Fig. 13 show that XQueC’s total decompression time grows linearly with the size of the data to be output, which, in this case, linearly depends on the document size. The \( \text{//homepage} \) query is very selective, while \( \text{//person} \) returns about \( 1/10 \)th of the document. Fig. 13 also shows that the compression configuration impacts data decompression time. The worst performing, among these in Fig. 13, is Na"iveHuffman; \( CH \) is better, \( G^2H \) further improves a little, and \( GH \) is best. The performance difference stems from two factors: the time to read compressed data from containers, and the time to decompress it, following the respective compression configuration. As the queries grow complex, the difference levels off because of the more important reconstruction effort.

Our second experiment studied the relative performance of the SortedOuterUnion and Reconstruct approaches for the same three decompression queries, on XMark documents up to 111 Mb. The Reconstruct outperforms the SortedOuterUnion for more complex queries such as \( \text{//person} \), by about 30%. For simpler queries such as \( \text{//address} \) and \( \text{//address} \), the difference levels off, since the reconstruction overhead is also lower. We conclude that Reconstruct is more robust, especially for decompressing (large parts of) the input documents; also, it uses much less memory than the SortedOuterUnion.

6.3.5 Compression and decompression time comparison. We now compare XQueC’s compression and decompression times with the corresponding times of competitor systems. We only compare with XMill itself to give a baseline. As already discussed above, XQueC compression/decompression times are not directly comparable with XMill/XGrind given that the latters are compression tools rather than systems. For instance, in XQueC we do not expect to execute compression as much frequently as in a compression tool, and similarly we do not expect to decompress as many numerous times. Henceforth, the times for XQueC compression/decompression should be considered under the above disclaimer.

We ran a set of experiments comparing compression/decompression of incremental sizes of XMark documents with those obtained on XMill. These experiments study XQueC scaleup with larger document sizes. Table VI presents the compression/decompression
Table VI. XQueC CT/DT.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Xmill CT(s)</th>
<th>XQueC CT(s)</th>
<th>Xmill DT(s)</th>
<th>XQueC DT(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark111</td>
<td>111MB</td>
<td>29</td>
<td>2388</td>
<td>4</td>
<td>603</td>
</tr>
<tr>
<td>XMark75</td>
<td>75MB</td>
<td>22</td>
<td>1804</td>
<td>3.5</td>
<td>456</td>
</tr>
<tr>
<td>XMark50</td>
<td>50MB</td>
<td>14</td>
<td>1183</td>
<td>2.5</td>
<td>300</td>
</tr>
<tr>
<td>XMark30</td>
<td>30MB</td>
<td>7.5</td>
<td>620</td>
<td>1.2</td>
<td>157</td>
</tr>
<tr>
<td>XMark15</td>
<td>15MB</td>
<td>4.3</td>
<td>356</td>
<td>0.7</td>
<td>96</td>
</tr>
<tr>
<td>Nasa</td>
<td>30MB</td>
<td>3.3</td>
<td>536</td>
<td>0.9</td>
<td>138</td>
</tr>
</tbody>
</table>

time of XQueC using NaïveHuffman \(^4\) and Xmill. The difference is quite important, and is mainly due to our storage procedure which is more involved and implies BerkeleyDB.

6.4 Query execution times

In this section, we provide further insight on XQueC’s general XQuery query evaluation performance. We examine several aspects of XQueC performance. Section 6.4.1 studies XQueC’s scaleup on complex queries on increasingly large document sizes. Section 6.4.2 compares XQueC query performance with other compressed XML databases. Finally, Section 6.4.3 demonstrates trade-offs between lazy decompression, as implemented in XQueC, and eager and transient decompression, which we implemented for the purpose of comparing. A comparison of variable binding performance in XQueC and in similar systems using a tag-partitioned structural index is delegated to the appendix (Section 8.1), together with a discussion on the possible reasons for such differences.

6.4.1 XQueC performance. We study the scaleup of XQueC’s query engine with various document sizes, and the impact of workload-oriented compression configuration on query execution times. To reflect the general case of queries which may construct arbitrary structure, we used the SortedOuterUnion approach for the QEPs of this section. The performance difference between Reconstruct and SortedOuterUnion-based plans was illustrated in the previous section.

We measured the XMark query $XQ_1$ on the XMark documents compressed with various compression configurations; the results are reported in Figure 14, for the configurations NaïveHuffman1 (as a baseline), and for the workload-oriented $CH$, $GH$ and $G^2H$. We notice that, as expected, query time scales linearly with the document size. We measured both hot and cold execution times. Fig. 14 shows that for each configuration, hot and cold runs are quite similar; this is because the execution is pipelined, and (for such small data volumes) dominated by CPU time, both for iterating over tuples and for decompressing. Overall, it can be seen that $GH$ has the best performance, followed by $G^2H$, $CH$, and finally NaïveHuffman.

We have performed similar measures with $XQ_8$ and $XQ_{14}$, and obtained overall similar results. We noticed a linear scaleup with the document size, even for a join query like $XQ_8$, confirming XQueC’s scalable performance due to its storage model and query engine. Furthermore, the configuration $GH$ provided the best results, followed by $G^2H$.

We close this section with a very simple measure, but which brings an insight to the impact of the compression configuration on query execution time. We measured the time to read and decompress all containers.

Fig 15(top) details this for two documents of similar size but different structure:

\(^4\)These times with other Naïve or cost-wise configurations look approximately the same; as explained, decompression costs decreases when using ALM.
Andrei Arion et al.

Fig. 14. XMark $XQ_1$ query evaluation time on various compression configurations. XMark17 and Shakespeare. We consider just two extreme configurations: NaïveHuffman1 and NaïveALM1, as well as the non-compressed scenario (whereas the time to decompress is simply the time to copy data from compressed words into strings).

Fig. 15(top) shows that character-based Huffman decompression can be quite slow when compared with ALM decompression. At the same time, thanks to a better compression ratio, the time to read data from disk is smaller for the NaïveALM1. Thus, the overall time is minimized by using ALM, which is better than the non-compressed scenario. This observation is the key to understanding the performance of various XQueC compression configurations presented in this paper, which have to choose between ALM and Huffman. ALM turns out to be quite heavily used by our heuristics; presumably, Huffman might be preferred if compression time was taken into account.

Decompression time is more important on the XMark document, when compared to Shakespeare. This can be explained by the fact that Shakespeare tends to have relatively short strings ("lines" have bounded length), while XMark’s more verbose types (such as item descriptions etc.) feature longer strings. The same trend is confirmed by the graph in Fig. 15(bottom), using larger documents. In this second graph, we also differentiate between hot and cold data access time. We show that the difference is not very important, also due to XQueC’s fine data access granularity, which makes the read cost dominated by the work to be done for each (small) data granule.

The overall effect of compression, and of compression configurations, on query performance can be summarized in light of this simple experiment (see also Section 6.4.3): (i) character-based decompression, such as Huffman, is slower than word-based decompression; (ii) a better compression ratio shortens the time it takes to read data from disk; and (iii) decompression time is more important in queries featuring text-rich data.

6.4.2 Comparison with other queryable XML compressors. As explained in Section 6.1.3, we could only obtain XGrind [Tolani and Haritsa 2002] but were unable to run queries on it, and the system described in [Buneman et al. 2003], which is, however, too different to yield meaningful comparisons. Thus, we do not have comparison results.

6.4.3 Lazy vs. eager vs. transient decompression. In this section, we study the impact of lazy decompression, and on compression in general, on query processing performance.

To that purpose, we compare the performance of two QEPs, depicted at top left in Fig. 16. Both QEPs perform an equality join among two text containers, on their (possibly compressed) text values. Furthermore, to vary the size of the join inputs, we add to one of the inputs a selection operator called FirstN, which examines its entire input, and eliminates all but the first N input tuples. We use FirstN as a surrogate for various-complexity
sub-query plans which may prune some tuples from the input.

Both QEPs in Fig. 16 implement the join as a HashJoin. The lazy QEP decompresses directly the join output; the eager one decompresses the containers just after reading them, possibly, in order to allow the execution in the decompressed domain of the complex sub-plan that \texttt{FirstN} represents here. Join inputs are decompressed by XQueC before joining two containers compressed differently (we decompress them here for the sake of the measure). For simplicity (and to have non-empty join results) we have considered the same container on both join branches. We use the \texttt{/play/act/scene/speech/line} container derived from the the Shakespeare document described in Tab. IV; this container holds most of the document's content, and has about 100,000 entries. We have taken \( N \) to be 10, 1,000, 10,000, respectively 50,000. The container was compressed using Huffman. The join result contains approximately \( N \) tuples, since there are very few duplicates in the container.

Furthermore, we have considered two alternatives: when the HashJoin optimizer correctly identifies its smaller input and builds it into the hash table, which we call \emph{small build}, and the opposite, when the optimizer is wrong and builds the larger operator into the hash table; we call the latter alternative \emph{large build}.

At the top at right in Fig. 16, we plot the memory consumption for the lazy and eager QEPs, with a small or large built table. Large build here means the whole container is built into the hash table, regardless of the selectivity of \texttt{FirstN}. The eager QEP will build tuples of IDs and decompressed strings into the hash table, while the lazy one will build tuples of IDs and smaller compressed strings. Fig. 16 shows that the space occupancy of the lazy plan is clearly smaller than the eager one’s. The same ratio is kept in the small build plan, although the hash tables are smaller in this case. This simple example shows that manipulating compressed strings actually reduces memory needs at runtime, even in the case of a memory-consuming operator such as a (memory-based) HashJoin.

At the bottom in Fig. 16, we plot the decompression time, respectively, the join time,
in the lazy and eager QEPs. The data access time (omitted) is 10 seconds in all cases. The decompression time of the eager QEP is also the same. The lazy decompression time grows linearly with N and thus the size of the output. The join time is the time to build and then probe the hash table. In both graphs at the bottom of Fig. 16, the eager join, which manipulates strings, is about 40% slower than the lazy one, which manipulates (the same number of) compressed strings. The join is slower when doing a large build.

Note that the time and space requirements of the HashJoin by itself in the eager decompression plan coincide with the performance attained in the absence of any compression. This demonstrates (as expected) that compression reduces the memory needs of operators such as a HashJoin, which can be beneficial even for small computations such as this one. Clearly, larger data sets could produce more important differences.

This experiment, featuring a rather selective join, demonstrates two desirable features of lazy decompression: low space requirements due to the manipulation of compressed values, and little decompression effort. If the join result size is smaller than the cumulated size of the inputs, lazy decompression is typically faster. If one replaces the HashJoin in Fig. 16 with a NestedLoopsJoin, eager decompression becomes even less attractive, since one of the entries will be decompressed multiple times.

In contrast, as soon as the join output size is larger than the size of the input, lazy decompression is slower (makes more effort). The trade-offs between lazy and eager decompression can be seen as a particular instance of ordering joins and expensive functions [Chaudhuri and Shim 1999]. Notice that if we allowed the decompress operator to use a cache, as in [Hellerstein and Naughton 1996], then the join input and output sizes are no longer relevant, the number of decompression only depends on the number of distinct values, and lazy decompression is always better. However, in XQueC, decompression does not seem to be sufficiently expensive to justify the overhead of a cache, especially one that may overflow on disk as in [Hellerstein and Naughton 1996].

Our choice in XQueC was to compress in a way that would allow lazy decompression,
and the optimizer currently uses lazy decompression only. In a more general view, statistics on XML data values, and/or a cache in the decompress operator, could be applied.

**Transient decompression.** We now discuss the interest of transient data decompression. Transient decompression has been shown to be useful in compressed relational databases; the idea is to decompress data temporarily e.g., in order to perform a join, then compress it again, and manipulate it compressed until the end of the QEP. The benefit comes from using less space for intermediary query results.

For transient decompression to differ from lazy decompression, several value comparisons on the same attribute(s) must appear in the plan. For transient decompression to be useful, memory-consuming operators must be used (not only pipelined). These conditions are met by the XMark query $XQ_9$; we measured it using HashJoins, with lazy, eager, and transient decompression. The results are similar to Fig. 16.

In general, in XML queries, use cases for transient decompression are likely to be rarer than in relational databases. This is because much of XML query complexity comes from navigating structure, which *de facto* replaces some of the operations that in relational databases required value joins.

### 6.5 Conclusion of the experiments

Our experiments have studied several aspects of the XQueC XML querying and compression platform: the choice of a compression configuration, the impact of the configuration on the compressed data size, and on the query performance. Overall, the $GH$ and $G^2H$ heuristics provide the most interesting and stable compression performance, which is not attained by naïve compression configurations.

We have demonstrated the scalability of the query engine based on XMark queries. XQueC’s performance is due to the combination of its compression strategy, vertically fragmented storage model, and efficient operators. A big advantage of XQueC comes from its selective data access methods, provided by its path-partitioned storage; this is further discussed and demonstrated in the Appendix (Section 8.1).

XQueC is overall slower, and a weaker compressor, than other queryable XML compressors. However, XQueC is the only system to adopt a full database-style approach, and thus to support complex queries in a scalable manner. Competitor systems do not use a persistent store as XQueC does (some do not even produce an actual compressed disk-resident document), and cannot handle even simple path queries if the dataset goes beyond the available memory size.

XQueC’s approach is to bring compression into an XML database. This brings an overhead, due to: the fine-grain data access employed in XQueC, the need to access a persistent store, and the usage persistent element identifiers (a need faced by all persistent XML stores). We have shown, however, that XQueC achieves reasonable reduction of the document’s structure and contents, all the while allowing full database-style query processing. Furthermore, in the line of real data management systems, XQueC strives to process queries with very low memory footprint, unlike its memory-based competitors.

### 7. CONCLUSIONS AND FUTURE WORK

The XQueC system described in this paper takes the approach of including compression in an XML storage and querying system. XQueC mediates between several performance aspects of XML queriable compression, such as: compression factor, compression dictionary...
size, query execution time, and decompression time. XQueC balances these factors by the use of a cost model to help chose the most appropriate compression scheme. Furthermore, XQueC uses a fragmented storage model allowing for efficient structural query processing, following overall a lazy decompression approach.

Future works will concern a more comprehensive characterization of the way the container similarities may be evaluated and of the costs $c(x)$ of the employed algorithms. Another direction of research will be that of studying the impact of our compression structures on full-text queries, as considered in recent work [Amer-Yahia et al. 2004]. Indeed, full-text queries [XQUE 2004] constitute a significant fragment of XQuery queries, which can be interestingly executed in the compressed domain.

ACKNOWLEDGMENTS

We gratefully acknowledge the contribution of our former students Gianni Costa and Sandra D’Aguanono to the prototype described in [Arion et al. 2004]. Last but not least, we wish to thank Erika De Francesco for her help on a new implementation of the ALM algorithm.

REFERENCES


XQueC: Embedding Compression Into XML Databases


8. APPENDIX
8.1 Query processing in XQueC vs. tag-partitioned XML databases

In this section, we compare variable binding performance, based on XQueC’s path partitioned model (PP), and on the alternative tag-partitioned model (TP) discussed in Section 5.4.

We denote by linear path expression (lpe) a path expression of the form $l_1(//l_2(//...l_k//))$. For generality, we base our discussion on $//$-connected lpes; a set of such expressions is shown in Tab. VII.

<table>
<thead>
<tr>
<th>Path</th>
<th>Data set</th>
<th>Path</th>
<th>Data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$ //item</td>
<td>XMark111</td>
<td>$P_2$ //description</td>
<td>XMark111</td>
</tr>
<tr>
<td>$P_3$ //bold</td>
<td>XMark111</td>
<td>$P_4$ //category</td>
<td>XMark111</td>
</tr>
<tr>
<td>$P_5$ //title</td>
<td>DBLP</td>
<td>$P_6$ //author</td>
<td>DBLP</td>
</tr>
<tr>
<td>$P_7$ //Descr</td>
<td>SwissProt</td>
<td>$P_8$ //description//parlist</td>
<td>XMark111</td>
</tr>
<tr>
<td>$P_9$ //parlist/listitem</td>
<td>XMark111</td>
<td>$P_{10}$ //regions/item//description</td>
<td>XMark111</td>
</tr>
<tr>
<td>$P_{11}$ //person/name</td>
<td>XMark111</td>
<td>$P_{12}$ //regions/item//listitem</td>
<td>XMark111</td>
</tr>
<tr>
<td>$P_{13}$ //category/name</td>
<td>XMark111</td>
<td>$P_{14}$ //article/title</td>
<td>DBLP</td>
</tr>
<tr>
<td>$P_{15}$ //book/author</td>
<td>DBLP</td>
<td>$P_{16}$ //Entry/METAL//Descr</td>
<td>SwissProt</td>
</tr>
<tr>
<td>$P_{17}$ //LIPID//Descr</td>
<td>SwissProt</td>
<td>$P_{18}$ //dblp/book/title</td>
<td>DBLP</td>
</tr>
</tbody>
</table>

Table VII. Linear path expressions of length 1 (top), 2 (bottom left) and 3 (bottom right).

We start by considering the problem of binding one variable to a lpe. The case most favorable to TP is a variable bound to a lpe of the form $//$tag: PP merges several IDScans, while TP only looks up the IDs of the given tag.

Fig. 17 compares the execution time for binding plans when the TP and PP plans differ.

The top graph corresponds to the lpes of length 1 in Table VII. The cardinalities of the binding plans range from 2,000 ($P_1$) to 720,000 ($P_7$). The PP overhead, due to scanning and merging different sequences, is negligible, since the Merge is pipelined and quite efficient. The maximum number of paths is attained for $//$bold, namely 99.

When binding a variable to longer lpes, TP is handicapped by its unselective data access. For example, consider binding a variable to the path $//$site/people/person/name: TP only allows to access all name IDs. Most of such IDs are not on the required path, and instead may be on paths 11, 18 etc. (Fig. 1), leading to useless data scans. To separate just the person names, TP accesses all person IDs (thus read even more data!), and applies a structural join between the two. Similarly, all IDs of elements labeled site and people IDs must be read and successively joined, to separate just the proper names.

In general, to bind a variable to a lpe of length $k$, TP must scan the IDs of all tags involved, and compute a $k$-way join. PP just scans the result. If ancestor IDs appear on related paths, a duplicate elimination step is also needed.
Optimized tag partitioning (OTP). One may wonder if TP binding performance could be improved by a static analysis of the lpes, which may identify: useless navigation steps, such as //person in the lpe //person/profile (only person have profiles), and needless sort and duplicate elimination steps, for every structural join plan that may compute the lpe.

In general, given an lpe, there may be several lpes consisting of fewer navigation steps, equivalent to the original one. We call optimized tag partitioning (OTP) the binding strategy with, for a given lpe, based on the path summary, enumerates all equivalent lpes, and picks the one with best performance on a tag-partitioned storage. Thus, OTP uses the cheapest combination of StackTreeDesc and StackTreeAnc, and only applies unavoidable sorting and duplicate elimination step. The intuition is that OTP provides the best binding strategy, based on a tag-partitioned structural index.

Fig. 17 (bottom) considers lpes of length 2 and 3 from Tab. VII. For P8 and P9, PP and OTP coincide; TP performs one extra join and a dup-elim. From P10 to P17, OTP only spares the dup-elim, which is cheap since the join result order was favorable. The performance difference between PP and OTP reflects the proportion of useless IDs scanned by OTP due to the tag-partitioned storage. This proportion reaches 430 for P14. PP is also faster on lpes of length 3, up to two orders of magnitude.

We conclude from these measures that PP outperforms TP, as well as OTP, for the task of variable binding. The difference is more important for longer path expressions, as TP scans potentially more IDs.
Combining variable bindings. Fig. 18 compares TP and PP performance when binding two, respectively, three related variables. In these cases, TP cannot benefit from any optimization, since the scans and the structural join are required, in order to get binding pairs. Furthermore, PP must also scan many inputs and join them. Also, in Fig. 18, PP and TP use the best possible join order.

In Fig. 18 (left), we consider a set of patterns $P_0^8$ to $P_0^{17}$, obtained from the patterns $P_8^1$-$P_1^7$ by binding a variable to each navigation step. For instance, while $P_8^1$ is //description//parlist, $P_0^8$ stands for the two-variable pattern “for $x$ in //description, $y$ in $x$//parlist”. We notice that for $P_0^8$ and $P_0^9$, PP and TP are very close, since: TP makes no useless reads (all parlist elements are under description elements, and similarly for $P_0^9$), and the cost is dominated by data access. We have isolated the join time in all these measures, and the biggest value was 4.2 seconds for $P_0^{17}$. Thus, the ID scans of TP translate directly into an important performance differences with respect to PP.

In Fig. 18 (right), we consider three-variable patterns. We have chosen just 1-tag $lpes$ to connect $x$ to $y$ and $z$, to pick the best possible case for TP: if longer paths connect variables, TP will do useless scans and joins, as previously explained.

Overall, scan, join, and total time are still smaller for PP than for TP. For $T_1$, $T_3$, $T_4$ and $T_6$, the join cost is negligible for both, compared to the data scan cost. In these cases, simply using a PP storage provides for faster binding. For $T_2$ and $T_5$, the joins by themselves take a more important part; even with the best ordering, the first join creates many intermediary results, on which the second join spends more time. This problem can be alleviated using holistic twig joins [Bruno et al. 2002], which reduce intermediary results in large structural join trees.

Measuring the difference between path and tag partitioning. A legitimate question arises: do these two partitionings differ significantly in general, or is the difference only noticeable in some contrived data sets and queries?
The answer depends on the number of different paths in document $d$ which end in a given tag $t$; we call this number the fan-in of $t$ in $d$ and denote $f_{in_{t,d}}$. If the fan-in of all tags in a given document is 1, then TP and PP coincide. The bigger the fan-in is, the more likely it is that TP and PP will perform differently when binding variables.

Considering the maximum (or the average) value of $f_{in_{t,d}}$, over all tags $t$ in a document $d$, does not account for the relative importance of each tag $t$ in $d$. Thus, we consider the median fan-in of $d$, defined as:

$$mf_d = \sum_{t \in d} f_{in_{t,d}} \frac{N_{t,d}}{N_d}$$

where $N_d$ is the number of (element and attribute) nodes in $d$, while $N_{t,d}$ is the number of nodes labeled $t$. This measure gives greater weight to tags present in greater number in the document. The median fan-ins for the documents in Table IV are shown below.

<table>
<thead>
<tr>
<th>Doc.</th>
<th>TreeBank</th>
<th>INEX</th>
<th>XMark111</th>
<th>SwissProt</th>
<th>Shakespeare</th>
<th>DBLP</th>
<th>NASA</th>
<th>UW</th>
</tr>
</thead>
<tbody>
<tr>
<td>recursion</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$max f_{t,d}$</td>
<td>49,901</td>
<td>1,722</td>
<td>99</td>
<td>39</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>$mf_d$</td>
<td>19.187</td>
<td>82</td>
<td>15.79</td>
<td>11.08</td>
<td>6.06</td>
<td>5.47</td>
<td>2.92</td>
<td>1.0</td>
</tr>
</tbody>
</table>

In all but the UW course data, $mf_d$ is quite important. In the UW course datasets, the maximum fan-in registered is 1, thus TP and PP coincide. Interestingly, this document has been produced from a relational data set, by a database Ph.D. student [UWXML 2004], who might have thought it proper to give distinct names only; this cannot be expected in general. The $max f_{t,d}$ and $mf_d$ values may be quite different, which justifies introducing the $mf_d$ measure. Let us analyze the possible reasons for large fan-ins.

Recursive elements, like XMark’s `parlist`, are one source, since the recursive element tag may appear on various paths. Recursion is actually encountered in a significant part of XML documents on the Web [Mignet et al. 2003].

Another source is the presence of common tags like name, text, `desc`, `@from`, which apply in different contexts, and thus appear on different paths, with different meaning.

A third source is XML flexibility, which may lead to a data annotation at the level of tags (metadata), or at the level of the values (data). For example, the XMark `item` data is split on 6 different paths, under the elements `europe`, `asia` etc.

Finally, textual XML documents tend to exhibit important fan-in values. Tags in such documents correspond to language components (e.g., “verb group”, “nominal group”) or markup elements (e.g., “italic”, “bold”), which can be arbitrarily nested. This phenomenon is present to some extent in the XMark documents, and is very visible in TreeBank and INEX. Such documents are representative for an important class of text-centered XML applications, arguably, of more interest for XML research than XML-ized relational sources. PP variable binding is likely to outperform TP on such data.

8.2 Comparison of path summaries and compressed XML trees

We could not compare XQueC’s CF with [Buneman et al. 2003], since that tool does not manipulate any disk-resident structure other than the XML file. Furthermore, it does not compress values, which, as we have shown, make an important part of a document’s contents. Instead, [Buneman et al. 2003] compresses the XML structure tree into a compact DAG, associating to each DAG node the set of corresponding XML element nodes. An interesting comparison is the number of nodes created in the path summary by top-down
merging in XQueC, denoted $N_{PS}$, and the number of nodes in the DAG of [Buneman et al. 2003], denoted $|DAG|$. The results are shown in Table VIII.

<table>
<thead>
<tr>
<th>Document</th>
<th>Shakespeare</th>
<th>XMark15</th>
<th>Nasa</th>
<th>Treebank</th>
<th>SwissProt</th>
<th>XMark111</th>
<th>DBLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>7.5 MB</td>
<td>15 MB</td>
<td>24 MB</td>
<td>82 MB</td>
<td>109 MB</td>
<td>111 MB</td>
<td>128 MB</td>
</tr>
<tr>
<td>$N_{PS}$</td>
<td>58</td>
<td>511</td>
<td>24</td>
<td>338,738</td>
<td>117</td>
<td>514</td>
<td>125</td>
</tr>
<tr>
<td>$</td>
<td>DAG</td>
<td>$</td>
<td>1,121</td>
<td>10,629</td>
<td>8,391</td>
<td>319,654</td>
<td>38,936</td>
</tr>
</tbody>
</table>

Table VIII. XQueC number of path summary nodes vs. the number of nodes in the DAG obtained by bisimulation.

Table VIII shows that XQueC’s path summary is generally smaller (in some cases by two orders of magnitude) than the DAG obtained by [Buneman et al. 2003]. This is explained by XQueC’s node equivalence notion, based on paths, generally weaker than the one of [Buneman et al. 2003]. The difference is striking, e.g., in the case of XMark data sets; the presence of recursive, variable and repeated structure in the data leads to a relatively large DAG, but a compact path summary. Going from a 15 MB to 111 MB one, the path summary adds 3 paths, but the DAG size has more than tripled.

For Treebank, the DAG is slightly smaller than the path summary (by less than 3%). However, where the work in [Buneman et al. 2003] stops by noticing that Treebank is not really compressible, XQueC applies robust encoding techniques to compress it to less than 45 MB, mostly by compressing its values (as shown in Fig. 9).