Optimizing XML Compression in XQueC

Andrei Arion[1]*, Angela Bonifati[2], Ioana Manolescu[3], Andrea Pugliese[4]
[1] SCORT SA, France
[2] ICAR CNR, Italy
[3] INRIA Saclay–Île-de-France, France
[4] DEIS – University of Calabria, Italy

Abstract

We present our approach to the problem of optimizing compression choices in the context of the XQueC compressed XML database system. In XQueC, data items are aggregated into containers, which are further grouped to be compressed together. This way, XQueC is able to exploit data commonalities and to perform query evaluation in the compressed domain, with the aim of improving both compression and querying performance. However, different compression algorithms have different performance and support different sets of operations in the compressed domain. Therefore, choosing how to group containers and which compression algorithm to apply to each group is a challenging issue. We address this problem through an appropriate cost model and a suitable blend of heuristics which, based on a given query workload, are capable of driving appropriate compression choices.

1 Introduction

XML compression has gained prominence recently because it counters the disadvantage of the ")5(verbose" representation XML gives to data. Our *XQueC* system is a full-fledged data management system for compressed XML data that addresses the problem of embedding compression into XML databases without degrading query performance [4, 5, 6, 7]. XQueC is capable of covering a significant fragment of XQuery while providing efficient query processing on compressed XML data.

The storage model we designed for XQueC leverages a proper data fragmentation strategy, which allows the identification of the units of compression (*granules*) for the query processor; these are also manipulated at the physical level by the storage backend. This fragmented model supports fine-grained access to individual data items, providing the basis for diverse efficient query evaluation strategies in the compressed domain. It is also transparent enough to process complex XML queries.

In the XQueC storage model, data nodes found under the same path are grouped into a single *container*. Containers are further aggregated into *groups*, which allow their data commonalities to be exploited, thus allowing both compression and querying to be improved. However, when deciding how to group containers, several other factors must be considered that impact the final compression ratio and the query performance. For instance, if we take into account decompression times, our choice of how to group containers should ensure that containers belonging to the same

*This work has been performed while the author was affiliated with the INRIA Saclay.

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1692

group also appear together in query predicates. In fact, it is always preferable to perform the evaluation of a predicate within the compressed domain; this can be done if the containers involved in the predicate belong to the same group and are compressed with an algorithm supporting that predicate in the compressed domain. XQueC addresses these issues by employing a cost model and applying a suitable blend of heuristics to make the final choice. In [4, 5, 7] we demonstrated the utility of XQueC by means of thorough experimental assessments on a variety of XML datasets and by comparing it with available competitor systems.

In this paper we briefly recall XQueC's storage model then present our approach to making a cost-based choice of the compression granules and corresponding compression algorithms.

2 Storing XML data in XQueC

XQueC splits an XML document into three data structures: a *structure tree*, a set of *containers* and a *structure summary*. Across all these structures, XQueC encodes element and attribute names using a simple binary encoding. The structure tree is encoded as a set of ID sequences, each associated with a different root-to-node path in the tree. To encode the IDs in all its storage structures, XQueC uses conventional *structural identifiers* consisting of triples [pre, post, depth] as in [1, 9, 10, 16]. The pre (post) number reflects the ordinal position of the element within the document, when traversing it in preorder (postorder). The depth number reflects the depth of the element in the XML tree. This node identification scheme allows the direct inference of structural relationship between two nodes using only their identifiers.

Similarly, the containers store together all data values found under the same root-to-leaf path in the document. A container is realized as a sequence of records, each consisting of a compressed value, and a number representing the position of its parent in the corresponding ID sequence of the tree structure.

Finally, the storage model includes a *structure summary*, i.e., an access support structure storing all the distinct paths in the document. The structure summary of an XML document d is a tree whose nodes uniquely represent the paths in d, that is, for each distinct path p in d, the summary has exactly one node on path p. For a textual node under path p, the summary has a node labeled $/p/\#\mathsf{text}$, whereas for an attribute node a under path p, the summary has a node labeled $/p/@a$. This establishes a bijection between paths in an XML document and nodes in the structure summary. Each leaf node in the structure summary uniquely corresponds to a container of compressed values.

Figure 1 outlines XQueC's architecture. The loader decomposes the XML document into ID sequences and containers, and builds the structure summary. The compressor partitions the data containers and decides which compression algorithm to apply (see Section 3). This phase produces a set of compressed containers. The repository stores the storage structures and provides data access methods and a set of compression functions working at runtime on constant values appearing in the query. Finally, the query processor includes a query optimizer and an execution engine providing the physical data access operators.

3 Optimizing compression configurations

The compression of data containers is definitely more efficient if appropriate container groups are considered and compressed together. There may exist multiple grouping choices, which have a

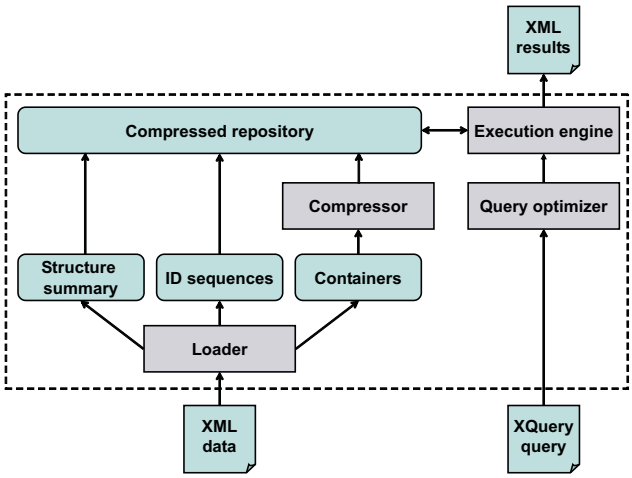

Figure 1: Architecture of the XQueC prototype

non-trivial impact on the size of compressed data and on the achievable query performance. As explained in the rest of this section, XQueC leverages a suitable cost model to drive the final choice.

3.1 Rationale for a cost model

The containers include a large share of the compressible XML data, i.e., the values, thus making proper choices about compressing them is a key issue for an efficient XML compressor [14]. Similarly to other non-queryable XML compressors, XQueC looks at the data commonalities to choose the container's compression algorithm. But how do we know that a compression algorithm is suitable for a container or a set of containers? In principle, we could use any eligible compression algorithm, but one with nice properties is of course preferable. Each algorithm has specific computational properties, which may lead to different performance depending on the data sets actually used and on their similarities. In particular, the properties of interest for our purpose were the *decompression time*, which strongly influences the query response times over compressed data, the *compression factor* itself, and the *space usage of the source models* built by the algorithm. In fact, a container can be compressed individually or along with other containers; in the latter case, a group of containers share the same source model (i.e., the support structures used by the algorithm for both compressing and decompressing data). Grouping containers might be convenient, e.g., when they exhibit high data similarity. Therefore, the space usage of the source model matters as much as the space usage of containers themselves and the decompression time; combining these three factors makes the choice even more challenging.

Besides the properties discussed above, each compression algorithm is also characterized by the supported selections and/or joins in the compressed domain. There are several operations one can perform with strings, ranging from equality/inequality comparisons to prefix-matching and regular expression-matching; we give 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, the Huffman algorithm supports

both equality selections and equality joins in the compressed domain. Same holds for ALM, Extended Huffman [15], Arithmetic [19] and Hu-Tucker [11].

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

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

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

The final choice of the algorithms to employ for the containers is driven by the predicates that are actually evaluated in the queries. The specific advantage of XQueC over similar XML compressors is that XQueC exploits query workloads to decide how to compress the containers in a way that supports efficient querying. Besides selection and join predicates, the cost model also takes into account top-level projections (i.e., those present in `RETURN` XQuery clauses), as they enforce the decompression of the corresponding containers. Query workloads have been already successfully employed in several performance studies, from multi-query optimization to XML-to-relational mappings [8, 17]. To the best of our knowledge, this is the first time they are employed for deciding how to compress data.

We now illustrate the multiple factors that influence the compression and querying performances by means of an example. Let us consider three containers, namely c_1, c_2 and c_3, whose size are 500KB, 1MB and 100MB, respectively. Assume that the workload features an inequality join between c_1 and c_2 and a prefix join between c_1 and c_3, whereas containers c_2 and c_3 are never compared by the workload queries (Figure 2(a)). To keep the example simple, we disregard top-level projections.

If we aim at minimizing only the storage costs (thus disregarding the decompression costs) among the multiple alternatives (i.e., keeping the containers separated versus aggregating them in all possible ways), we would prefer to compress each container separately (Figure 2(b)). Indeed, making groups of containers often increases both the sizes of compressed containers and source models, because of the decreased inter-containers similarity within each group. In fact, if for instance c_1 and c_2 contain strings over two disjoint alphabets of two symbols each, and two separate source models are built, c_1 and c_2 are likely to be encoded with one bit per symbol. If instead a single source model is used, two bits per symbol are required, thus degrading the compression factor. A second relevant decision to be made is that of choosing the right algorithm for each separate container. Since only the storage cost matters, this algorithm should be the one with the best compression factor.

In contrast, if we aim at minimizing only the decompression costs, but keeping the advantage of the reduced amount of data to be processed, then we would have to find a compression algorithm that supports both inequality and prefix joins in the compressed domain. If such an algorithm is available, the best choice is the one that aggregates all containers into one group, compressed

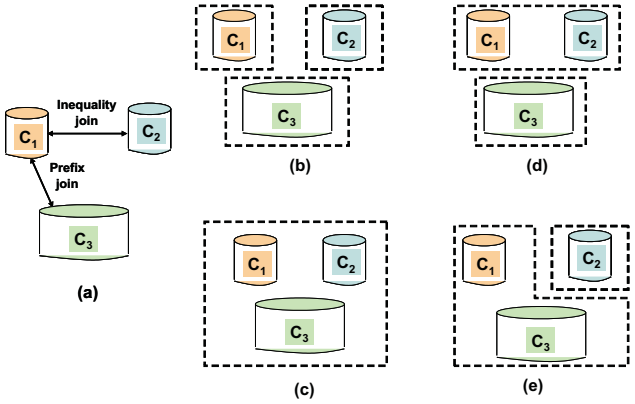

Figure 2: Sample workload and possible partitioning alternatives

with that algorithm (Figure 2(c)). Such a choice is optimal as it would nullify the decompression cost. Note that this is already in conflict with the above choice of minimizing only the storage costs. If instead such an algorithm is not available, and there is one order-preserving algorithm for inequality joins and a prefix-preserving one for prefix joins, two possible alternatives arise: grouping c_1 together with c_2 and compressing them with the order-preserving algorithm, leaving c_3 as a singleton; or, grouping c_1 together with c_3 and compressing them with the prefix-preserving one, leaving c_2 as a singleton. The first choice saves decompression of a very large container, i.e., c_3, thus making it preferable (Figure 2(d)).

The most general case is that of minimizing *both* storage and decompression costs. For the containers above, there are again many possible alternatives. If the prefix-preserving algorithm matches the one that minimizes the storage costs, the choice of grouping is straightforward – leaving c_2 as a singleton (Figure 2(e)). On the other hand, if the two algorithms do not match, or if the largest container is c_2, the scenario becomes increasingly more complex.

3.2 Costs of compression configurations

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

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

- the *decompression cost* $a.c_d(F)$ is a function estimating the cost of retrieving an uncompressed symbol from its compressed representation using algorithm a;
- the *storage cost* $a.c_s(F)$ is a function estimating the average cost of storing the compressed representation of a symbol using a;

- the *source model storage cost* $a.c_x(F, \sigma)$ is a function estimating the cost of storing the auxiliary structures needed to represent the source model of a set of containers sized σ using a;
- the algorithmic properties $a.\mathcal{L}$ are the kinds of comparisons supported by a in the compressed domain.

Note that each cost component is a function of the similarity among the containers. This is due to the fact that such costs always depend on the nature of data enclosed in the containers compressed together, i.e., on the similarity among them (see the example in the previous section). Observe also that, as opposed to the containers storage cost, the source model storage cost is not symbol-specific, but it refers to an entire source model. This is due to the fact that the overhead of storing the source model is seldom linear with respect to the container's size [15].

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

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

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

Summarizing, the cost model input consists of (see Table 1 for the symbols used):

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

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

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

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

Table 1: Summary of symbols used in the cost model

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

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

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

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

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

The decompression cost is evaluated by summing up the costs associated with both comparisons and projections in $\mathcal{W}$. To give an intuition, let us first consider a generic comparison occurring between two containers c_i and c_j. The associated decompression cost is zero if c_i and c_j share

the same source model and the algorithm they are compressed with supports the required kind of comparisons in the compressed domain. A non-zero decompression cost occurs instead when one of the following conditions holds:

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

For a selection over a container c_i, a zero decompression cost occurs only if the compression algorithm for c_i supports the required kind of selection in the compressed domain. In such a case, the constant value will be compressed using c_i's source model and the selection will be directly evaluated in the compressed domain. If instead the compression algorithm for c_i does not support the selection in the compressed domain, a non-zero decompression cost must be taken into account. To formalize this, we define a function d_{comp} that, given a compression configuration, calculates the cost of decompressing pairs of containers or single containers, when involved in selections. The pseudocode for function d_{comp} is shown in Figure 3, where function $set(P, c)$ returns the set in P containing c.

```
   function d_comp(s: compression configuration,
        i ∈ {1, ..., |C|} and j ∈ {0, ..., |C|}: container indexes,
        l ∈ L: comparison type): return a decompression cost
1    If j ≠ 0 // join predicate
2      p' ← set(P, c_i); p'' ← set(P, c_j)
3      If p' ≠ p'' Or l ∉ alg(p').L
4        Return |c_i| * alg(p').c_d(F_p') + |c_j| * alg(p'').c_d(F_p'')
5    Else // selection predicate
6      p ← set(P, c_i)
7      If l ∉ alg(p').L
8        Return |c_i| * alg(p).c_d(F_p)
9    Return 0
```

```
   function d_proj(s: compression configuration,
        i ∈ {1, ..., |C|}: container index): return a decompression cost
1    p ← set(P, c_i)
2    Return |c_i| * alg(p).c_d(F_p)
```

Figure 3: Decompression cost for comparison predicates and top-level projections

Similarly, function d_{proj}, given a compression configuration, calculates the decompression cost associated with the top-level projection of a container (Figure 3).

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

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

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

3.3 Optimizing compression choices

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

Greedy heuristic. We have devised a greedy heuristics that starts from a naive initial configuration, s_0, and improves over it by merging sets of containers in the partition. The main idea here is that of exploiting each comparison in $\mathcal{W}$ to enhance the current configuration; at each iteration, the heuristic picks the comparison that involves the maximum number of containers (improving over the heuristic presented in [5] that randomized the choice of the comparison). Figure 4 shows the pseudocode of this heuristic. Steps 1 to 19 build the initial configuration by examining all the comparisons in the workload. Then, steps 20 to 32 examine the cost of possible new configurations that are built by merging the groups obtained in previous steps but using a different algorithm for them. The algorithm halts when all comparisons in the workload have been inspected.

Group-based greedy heuristic. The group-based greedy heuristic is a variant of the greedy one, and relies on the simple intuition that textual data marked by the same tag will likely have similar text content. Indeed, this heuristic treats groups of containers corresponding to paths ending with the same tag as a single container; this may lead to the building of a less trivial initial configuration than the one produced by the greedy heuristic. The latter is eventually applied on this initial configuration; thus, the pseudocode looks like the one in Figure 4, except for the pre-processing step.

Clustering-based heuristic. Since the problem of computing the compression configurations can be also thought of as a clustering problem, we designed a heuristic that employs a simple clustering algorithm, i.e., the *agglomerative single-link* algorithm [13]. In our case, the distance between pairs of containers must reflect the costs incurred when compressing those containers with different algorithms. This cost, in turn, depends on the containers' actual content. In particular, the distance between containers is proportional to the cost for decompressing the containers and storing them and their corresponding auxiliary structures. Moreover, for each algorithm, a non-null decompression cost occurs whenever the two compressed containers are involved in comparisons not supported by that compression algorithm (in the compressed domain). The distance can thus be formalized as follows:

```
     function Greedy(W: query workload): return a compression configuration
1      W' ← W; s_0 = ⟨P_0, alg_0⟩
2      Repeat
3        c_i, c_j ← containers having the maximum number of comparisons in W'
4        W' ← W' \ {comparisons involving both c_i and c_j}
5        If ∄p ∈ P_0 | c_i ∈ p  or c_j ∈ p
6          add the set p^n = {c_i, c_j} to P_0
7          W ← W \ {comparisons involving both c_i and c_j}
8          A ← set of algorithms capable of doing the maximum number of comparisons
9                between c_i and c_j in the compressed domain
10         If |A| = 1
11           a ← the algorithm in A
12         Else
13           a ← the algorithm in A minimizing the expression
14                 α · a.c_d(F_{p^n}) + β · a.c_s(F_{p^n}) + γ · a.c_x(F_{p^n}, Σ_{c∈p^n} |c|)
15         Make alg_0 associate p^n with a
16     until W' = ∅
17     For each container c | ∄p ∈ P_0, c ∈ p
18       P_0 ← P_0 ∪ {c}
19       Make alg_0 associate {c} with an algorithm a chosen as at line 8
20     s_curr ← s_0
21     Repeat
22       pred ← predicate in W having the maximum number of occurrences
23       c_i, c_j ← containers involved in pred
24       p' ← set(P, c_i); p'' ← set(P, c_j)
25       P' ← P_curr \ p' \ p'' ∪ {p' ∪ p''}
26       For each a_i ∈ A
27         alg_{a_i} ← alg_curr
28         Make alg_{a_i} associate p^u with a_i
29         s_{a_i} ← ⟨P', alg_{a_i}⟩
30       s_curr ← argmin_{s∈{s_curr, s_{a_1}, ..., s_{a_|A|}}} cost(s)
31       W ← W \ { comparisons involving two containers in p^u}
32     until W = ∅
33     Return s_curr
```

Figure 4: Greedy heuristic

```
   function Clustering(W: query workload): return a compression configuration
1    dist_min, dist_max ← minimum and maximum distances among two containers in C
2    Divide the range [dist_min, dist_max] into equally-sized sub-ranges
3    For each sub-range r
4      If ∃ containers c_i, c_j | dist(c_i, c_j) ∈ r
5        P ← partition of C where containers c_i, c_j are in the same set
             only if dist(c_i, c_j) is less or equal to the lowest value in r
6        For each p ∈ P
7          Make function alg associate p with algorithm a
               that minimizes α · a.c_d(F_p) + β · a.c_s(F_p) + γ · a.c_x(F_p, Σ_{c∈p} |c|)
8        s_curr ← argmin_{s ∈ {s_curr, ⟨P,alg⟩}} cost(s)
9    Return s_curr
```

Figure 5: Clustering-based heuristic

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

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

The pseudocode of the clustering-based heuristic is reported in Figure 5. 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 chosen number of levels stems from a trade-off between execution times and probabilities of finding good configurations. Deciding the number of levels is empirically done, and implies some manual tuning, which is not required in the other heuristics. Finally, for each generated partition, the heuristic assigns to each set in the partition the algorithm that locally minimizes the costs.

4 Conclusions

In this paper we presented our approach to the problem of optimizing compression choices in the context of the XQueC compressed XML database system. By choosing how to group containers and which compression algorithm to apply to each group, XQueC is able to exploit data commonalities and to perform query evaluation as much as possible in the compressed domain. This improves both compression and querying performance. We addressed the problem of optimizing compression in XQueC through an appropriate cost model and a suitable blend of heuristics which, based on a given query workload, are capable of driving appropriate compression choices.

References

[1] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International*

Conference on Data Engineering, pages 141–152, San Jose, CA, USA, March 2002. IEEE.

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

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

[4] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing Queries to Compressed XML Data (demo). In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 1065–1068, Berlin, Germany, 2003. Morgan Kaufmann.

[5] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient Query Evaluation over Compressed XML Data. In *Proceedings of the International Conference on Extending Database Technologies*, pages 200–218, Heraklion, Grece, 2004.

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

[7] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Xquec: A query-conscious compressed xml database. *ACM Trans. Internet Techn.*, 7(2), 2007.

[8] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering*, pages 64–76, San Jose, CA, USA, 2002. IEEE.

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

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

[11] T. C. Hu and A. C. Tucker. Optimal Computer Search Trees And Variable-Length Alphabetical Codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, 1971.

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

[13] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

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

[15] E.S. De Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and Flexible Word Searching on Compressed Text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[16] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER:A Native System for Querying XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 672, San Diego, CA, USA, 2003. ACM.

[17] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 249–260, 2000.

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

[19] I. H. Witten. Arithmetic Coding For Data Compression. *Communications of ACM*, pages 857–865, 1987.