Column-oriented Database Systems and XML Compression

Tyler Corbin[1], Tomasz Müldner[1] and Jan Krzysztof Miziołek[2]
[1]*Jodrey School of Computer Science, Acadia University, Wolfville, B4P 2A9 NS, Canada*
[2]*Faculty of Artes Liberales, University of Warsaw, Warsaw, Poland*

Keywords: Column-oriented Databases, XML, XML Compression, XML Databases.

Abstract: The verbose nature of XML requires data compression, which makes it more difficult to efficiently implement querying. At the same time, the renewed industrial and academic interest in Column-Oriented DBMS (column-stores) resulted in improved efficiency of queries in these DBMS. Nevertheless there has been no research on relations between XML compression and column-stores. This paper describes an existing XML compressor and shows the inherent similarities between its compression technique and column-stores. Efficiency of compression is tested using specially designed benchmark data.

1 INTRODUCTION

The eXtensible Markup Language, XML (XML, 2013), is one of the most popular data encoding and exchange formats for the storage of large collections of semi-structured data, including scientific databases such as the Universal Protein Resource (The UniProt Consortium, 2013), OpenStreetMap (OSM) (OSM, 2014) a collaborative project to create a map of the world, Wikipedia (Wikipedia: The Free Encyclopedia, 2014), and even bibliographic databases, such as DBLP (The DBLP Computer Science Bibliography, 2014). Some of these systems use relational DBMS back-ends and are referred to as *XML Databases* (not to be confused with XML Database *engines*, which use XML documents as the fundamental unit of storage). The advantage of using DBMS back-end includes the support for ACID, scalability and transaction control but extends to XML-related functionality, e.g., updates (Müldner et al., 2010) and parallel implementations (Müldner et al., 2012).

For XML *documents*, the verbose nature of XML to support human readability may result in very large datasets (in hundreds of gigabytes), necessitating data compression. The compression rate can boosted by using XML *conscious* compressors, i.e., compressors aware of specific syntactical features. In various applications of XML, not only efficient compression/decompression is needed, but also efficient implementation of XML queries.

Column-Oriented Database Systems (*column-stores*) have seen a resurgence in academia and industry. A column-store is a relational database, with each attribute of a table (often represented as a column) stored in a separate file (or region in storage). If pages are laid out horizontally then to answer queries there is a lot of additional and unnecessary data being brought into memory, including all the columns, while in column-stores only columns that are necessary are fetched. In addition, storing data in columnar fashion increases the similarity of adjacent records on disk, and so provides the opportunity for compression; for other benefits see (Abadi et al., 2013).

This paper examines the inherent relationship between many types of XML-conscious compressors and column-stores, and shows that XML compressors and column-store are trying to solve very similar architectural issues with respect to storage and retrieval in the columnar environment. A specific example of an XML-conscious compression system, called XSAQCT is presented, see (Müldner et al., 2009) and see (Müldner et al., 2014) for theoretical background. XSAQCT requires the same functionality as a column-store, i.e., using similar path-based compression that resembles column-based compression in column-stores (while ignoring things such as SQL Joins).

Contributions. There are two main contributions of this paper: (1) the discussion of the relationships between XML compression and column-stores; (2) the analysis of results of testing the compression ratio using special benchmark data in form of an

Corbin T., Müldner T. and Miziołek J..
Column-oriented Database Systems and XML Compression.
DOI: 10.5220/0004995001070115
In *Proceedings of 3rd International Conference on Data Management Technologies and Applications* (DATA-2014), pages 107-115
ISBN: 978-989-758-035-2

XML corpus consisting seven files with various characteristics (for results of tests showing efficiency of queries see (Fry, 2011) and (Müldner et al., 2010)).

Organization. This paper is organized as follows. Section 2 describes related work in the area of column stores, XML compression, and a basic features of an XML compressor XSAQCT. Section 3 discusses the relationships between column stores and XML compression, and presents modifications of XSAQCT to satisfy these relationships, and Section 4 provides an overview of our architecture, the benchmark data, experimental results of testing, and analysis of results. Finally, Section 5 gives conclusions and future work.

2 RELATED WORK

2.1 Column-stores

There is a large body of research on using a standard relational DBMS as a back-end for the XML DBMS, e.g., (Florescu and Kossmann, 1999). In general, XML elements are stored in relations, while XML attributes, parent/child and sibling order information are stored as attributes. By *vertically partitioning* a relational database into a collection of columns that are stored separately, queries can read only the these attributes that are needed, rather than having to read entire rows and discard unneeded attributes only when they are in memory. We focus on *native* column-stores providing both a columnar storage layer and an execution engine (integrated with a traditional row-oriented execution engine), tailored for operating on one column-at-a-time with late tuple reconstruction (for joins and multi-selects), rather than column-stores with columnar storage only, recombining tuples automatically when brought into memory.

In the past few years, a number of industrial solutions have been introduced to handle workloads on large scale datasets stored using a Column-stores, for example: Redshift (Amazon, 2014), CStore (Stonebraker et al., 2005), VectorWise (Zukowski et al., 2012), and IBM DB2 (Raman et al., 2013). For a column-store to achieve performance similar to traditional row stores, column-store architectures (Abadi et al., 2013) generally apply Vectorized Processing, Late Materialization, and Compression.

2.2 XML Compression

This section first recalls various types of XML compressors and then describes their use to support queryability and updates. By an "XML Compressor", we always mean an XML-conscious compressor.

XML compressors are called *queryable* if queries can be processed with minimal or no decompression, otherwise they are non-queryable. XML compressors are called *permuting*, if the *structural part*, i.e., its markups and the *data part* of the XML document are separated during the encoding process. Finally, XML compressors can be classified based on the availability of an XML schema, (XML Schema, 2013) as *Schema-based compressors*, for which both the encoder and decoder must have access to the shared schema; and *Schema-less compressors*, which do not require the availability of the schema. For schema-less compressors, it is difficult to take advantage of specific compression techniques that work only on specific data types and allow direct operations on the compressed data. An absence of an XML Schema implies that there is no direct information indicating the data type, thus we have to assume that each data type is a CLOB (Character Large Object). Furthermore, it becomes even more difficult to take advantage of specific compression techniques that work only on specific data types and allow direct operations on the compressed data (e.g., a Run-Length encoding on Integers).

Formally, given an input document D, a permuting XML compressor $P_x(D)$ outputs a triple $<P,M,S>$, where P represents the model of the XML structure, S represents the storage of XML contents (often represented as a set of distinct containers, i.e., $S = \{S_1, \ldots, S_n\}$), and M maps elements in P to elements in S ($M : P \rightarrow S$). With respect to data compression, the goal of the mapping M is to use information from the model P to organize the storage S containers to reduce information entropy as much as possible; often by exploiting the mutual information defined by the partitioning strategy used by M. A variety of encoding schemes can be applied to the storage S, ranging from lightweight compression schemes operating on a sequence of values, e.g., Run Length Encoding (RLE), to heavyweight ones (compression on an array of bytes).

Finally, the model P has to be compressed/encoded using some model-dependent technique (e.g., tree compression).

A schema-less permuting XML compressor can approximate schema; the distinct elements of the model P can represent the constraints governing the structure of elements, and the type of CLOBs stored in each container $S_i \in S$ can be further approximated (e.g., a container with only characters between $[0,9]$ implies integer data type).

Regarding querying, XML queries are typically described using XPath (XPath, 2013) or

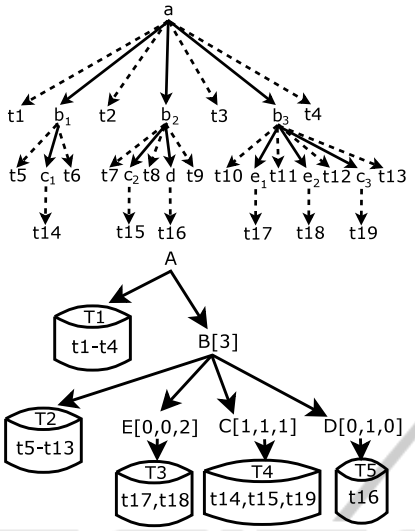

Figure 1: A document tree D (top) and its annotated tree (bottom).

XQuery (XQuery, 2013), which supplements XPath with a SQL-like "FLWOR expression" for performing joins and data-dependent operations. Using a single file to store the compressed XML document is a simple and convenient storage method, e.g., to satisfy any query, the model P would first be decompressed and traversed. Since P is often significantly smaller than the original XML file, the time penalty can be amortized over many queries, especially in comparison to homomorphic DFS-based query algorithms by smartly structuring the storage S. (A *homomorphic compressor* preserves the structure of the input). After the query is resolved in P, which may require the extraction of text elements, the appropriate text is found in S and decompressed to complete the query.

Regarding updates, the XQuery Update Facility (XQuery Update, 2013) is a relatively small extension of the XQuery language, which provides convenient means of modifying XML data.

Without substantial modifications to the compressed unit, all updates must be appended to the end of the file, Therefore the mapping M cannot be *fixed*, and must be able to integrate with some differential data structure to store updates which then is used to efficiently handle subsequent queries. Otherwise we will have to rebuild the model P and the storage S after each update (or flush) operation.

2.3 XSAQCT

XSAQCT is a permuting XML compressor, with its compressed representation (i.e., P) in the form of an Annotated Tree, see (Müldner et al., 2009).

An *annotated tree* of an XML document is a tree in which every node is represented by *similar paths* (i.e., paths that are identical) merged into a single path, and labeled by its tag name. Examples of similar paths are $/a/b_1/$ and $/a/b_2/$, or using XPaths $/a/b[1]/c[1]/$ and $/a/b[2]/c[1]/$. Each node of an annotated tree is of the form $X[A_0,...,A_n]$, i.e, it is labeled with a sequence (of non-negative integers), called an *annotation list*, representing the number of occurrences of this node's children. Figure 1 provides a running example of an XML document and its annotated tree. One of the necessary (but not sufficient) conditions for an annotated tree T to faithfully represent at least one arbitrary XML document D is that the sum of all annotations of a node is equal to the number of annotations of any child of this node, i.e., it satisfies the **recursive sums property**: For any node $X[A_0,...,A_n]$ in T, and any child of X, $Y[B_0,...,B_m]$, $\sum_{i=0}^{n} A_i = m$.

To answer queries using an annotated tree, annotations have to be mapped to the XPath indices, we now provide two algorithms. Algorithm 1 describes a SAX-event driven DFS traversal of the annotated tree, e.g., $SAX :: StartEvent()$ represents a handler called when the *StartEvent* event occurs (this algorithm ignores text elements, however to handle these elements, all that needs to be considered is the full-mixed content property and whether a node is a leaf or not). Algorithm 1 helps us to formalize how zero and non-zero annotations "ripple" through a DFS. Algorithm 2 describes how to pre-compute much of the DFS using highly-parallel, tightly-looped, recursive sums, showing how a regular XPath query can be resolved by an annotated tree.

Let a *cycle* be defined as consecutive siblings of the form $x \rightarrow y \rightarrow x$ or contradicting sibling orderings such as $x \rightarrow y, y \rightarrow x$ for different parents of the same similar path. While this algorithm is restricted to cycle-free XML sources, to handle cycles the only changes required are: annotation sequences need to be used instead of single annotation values, and the Line 5 has to be changed from a single value to a set of values, whose length is defined by an extra function parameter (the annotation of a "cycle" node).

Line 14 in Algorithm 1 says that after recurring on N's child X, we disregard the first element of X's annotation list for subsequent iterations and the first j elements of $X's$ annotation list will be disregarded by the time we decrement (in Line 15) j to zero. Thus, the size of X's annotation list is equal to the number of j decrements for N, which must be equal to the summation of N's annotation list (i.e., the recursive sum property). Generalizing this to find the proper

Algorithm 1: Annotated DFS.

```
   Data: struct Node = {
           ElementLabel P,
           ChildSet⟨Node⟩ S = {S_0,...,S_m}
           AnnotatedIterator A = {A_1,...,A_n}
   }
 1 AnnotatedDFS(Node N) begin
 2     j ← N.A_1
 3     if j < 1 then
 4         return
 5     else if N is attribute then
 6         for k ← 1 to j do
 7             SAX::Attribute(N.P)
 8     else
 9         if N is element Node then
10             SAX::StartEvent(N.P)
11         while j > 0 do
12             for every child X of N do
13                 AnnotatedDFS(X)
14                 X.A ← X.A.next()
15             j ← j − 1
16         if N is element Node then
17             SAX::EndEvent(N.P)
```

Algorithm 2: Random Access DFS.

```
   Data: struct Node = {
           SimilarPath P,
           ChildSet⟨Node⟩ S = {S_0,...,S_m}
           AnnotatedList A = {A_0,...,A_n}
   }
   struct XPath = {
           Sequence⟨Node⟩
           S = {S_j[X]/P.S_{j+1}[Y]/.../S_m[Z]}
   }
   Result: Returns true if XPath P can be resolved by
           an Annotated Tree rooted at N.
 1 RecursiveSum(Node N, XPath P, int offs) begin
 2     if |P.S| = 0 or offs ≥ |N.A| then
 3         return false
 4     pathVal ← X
 5     annotVal ← N.A_offs
       // Have to test all potential
          subtrees.
 6     if pathVal is ∅ then
 7         if |P.S| = 1 then
 8             return annotVal > 1
 9         bool c = false
10         for i ← 1 to annotVal do
11             SS = {S_j[i]/.../S_m[Z]}
12             c = c or RecursiveSum(N, SS, offs)
13         return c
       // Correct annotation index.
14     else if annotVal ≥ pathVal then
15         if |P.S| = 1 then
16             return true
17         disp = pathVal − 1
18         offset ← Σ_{j=0}^{offs−1} N.A_j
19         for every child Node X of N do
20             if X = P.S_{j+1} then
21                 SS = {P.S_{j+1}[Y]/.../P.S_m[Z]}
22                 return
                   RecursiveSum(X, SS, offset + disp)
23     return false
```

annotation locations for each vertex in some arbitrary path $V_0[X]/V_1[Y]/.../V_N[Z]$ in the annotated tree, we need to find how many times Line 14 will be executed for each vertex $V_j[]$.

The importance of the annotated tree in XSAQCT is that it can be considered a *high level XML index*, which has proved to be essential for various applications, e.g., updates, see (Müldner et al., 2010).

A lossless annotated representation of an XML tree efficiently supports many of the XQuery hierarchal queries, e.g., ancestor and descendant, just by comparing annotation values or sums for related uses).

The complete compression process involves creating *text containers* for each unique path of an XML document, storing a delimited (using ASCII Zero, \0), and possibly indexed list of *character data* for each similar path. Character data is a general term for all the characters not defined in the syntax of XML, e.g., text elements and structure (whitespace) data. For example, in Figure 1 the text container of `C[1,1,1]` has three text elements t_{14}, t_{15}, and t_{19}. Similar paths often have similar data (i.e., high mutual information), which improves the compression efficiency.

3 ARCHITECTURE OVERVIEW

3.1 XML Compression and Column-stores

(Choi and Buneman, 2003) consider a schema-based compressor XMill (Hartmut and Suciu, 2000) as a column-based storage model, and then study XML join queries. With respect to queryability, the goal of the mapping M, while working in the compressed domain, is to structure the model P in such a way to take advantage of vectorization and CPU- and cache-friendly operations, by creating a layout based on

fixed-width dense arrays, and thus being able to work only on the relevant data at a time. Permuting XML compressors are most useful, but they should also allow random access. While in general, a schema is not necessary for XML compression, for relational databases, it is always beneficial for a columnar database to exploit a priori knowledge from the schema (e.g., integer columns will be indexed and compressed using integer-specific techniques).

3.2 Discussion and Architecture Layout

The main claim made in this paper is that any permuting XML compressor that is to be extended to the transactional (i.e., update and query) domain, requires the same model of storage as a column-store (while ignoring additional nuances such as SQL Joins), as both have similar foundations in exploiting mutual information within disjoint columns, ultimately reducing the amount of time spent performing I/O operations. The justification for this claim is that any permuting XML compressor creates independent containers and then requires logic (i.e., the mapping M) to recombine the data from multiple locations. Similarly, a column-store often stores the attributes of an entity in several locations and then requires logic to recombine the data from multiple locations. Finally, since SQL substantially overlaps FLOWR, many of the complex data/storage dependent queries required for XQuery are identical to SQL queries in the relational domain.

For example, C-Store (Stonebraker et al., 2005) uses an instance of a BerkelyDB Key-Value database for each column. Several values are combined to produce blocks, which are compressed (depending on the data type, and whether direct operation on the datum is necessary) and finally stored into the database, using a sparse index on the position (i.e., key of the first instance in the block). On top of this abridged explanation of its architecture, C-Store introduces mechanisms for late materialization (e.g., using bit vectors or position lists to store intermediate query results making it possible to delay the decompression of data as long as possible), inserts/updates/deletes, and other SQL-specific functionalities.

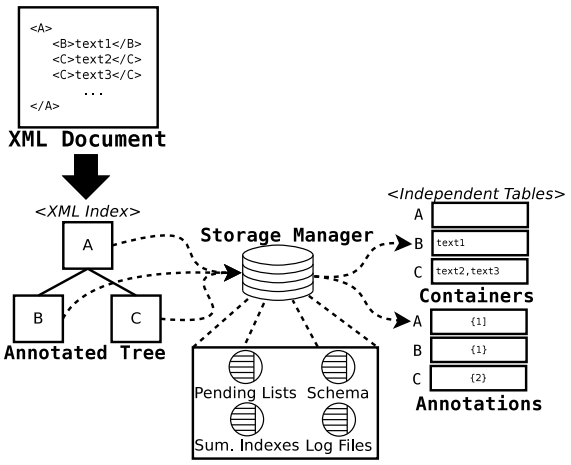

Figure 2: XSAQCT Storage Layout.

In the case of XML, it is difficult and sometimes incorrect to view the resulting tree as a collection of tuples. A single parent may have multiple children of the same name or type or that same node may only appear in a small percentage of subtrees rooted at the same parent. In recent years many XML-enabled (i.e., interfaces with a relational database), and Native XML solutions have been proposed. For example, eXist (exi, 2014) stores the XML tree as a modified, number-scheme based, k-ary tree combined with structural, range and spatial indexing based on B+-trees, and a cache used for database page buffers, but it does not compress the documents.

One of the main benefits of a permuting XML compressor is that the intermediate representation is compact. In addition, the permutation implies that we have two seemingly disjoint problems (compressing the structure and storing and compressing the content) that must be linked together via some mapping. In column-stores, it is beneficial to use column-oriented compression algorithms that can be operated on directly without decompression. Reading less data saves I/O operations, while not having to pay the cost of decompression. Similarly, by compressing the XML tree in a manner that can still be directly operated upon (e.g., we can still execute a DFS on the annotated tree to map it to text), we can introduce an ultra-compact index of the XML that can take advantage of the reduced I/O operations and other benefits from compressing the data.

With XSAQCT, if we view the contents of each "similar path" as an independent list of text values, we could create a single attribute table for each path in the relational domain; (either in the row-store or column-store environments). However, we can still exploit all of the benefits inherent to a column-store (compression, late materialization, etc.). Additionally, in the column-store environment, the contents of a single entity is often stored in many locations, which then requires additional logic to combine the attributes for joining and grouping attributes; (this is exactly how many permuting XML compressors work). Also, in (Fry, 2011) and (Corbin et al., 2013) XSAQCT was compared to many different XML-database engines using the BerkelyDB Key-value database, and the results were promising.

Figure 2 depicts the architectural layout of a modified XSAQCT. The storage manager is defined as an interface that relates the annotated tree and its querying mechanisms to the underlying storage architecture (either through SQL, or its own internal querying system). As discussed before, additional data structures are required to handle XSAQCT specific functionalities independent of the storage mechanisms involved (although the summation indices, for example, can be tightly integrated). It should be noted however, the memory required by these XSAQCT-specific features take away from the potential memory used by the caching, buffering and loading architectures of the database.

4 TESTING AND ANALYSIS

4.1 Benchmark Data

Table 1 provides our benchmark data; a corpus of seven XML files ranging from 30 Megabytes to 467 Gigabytes found in: (OSM, 2014), uniprot_trembl (The UniProt Consortium, 2013), (Wikipedia: The Free Encyclopedia, 2014), (The DBLP Computer Science Bibliography, 2014), (xmlgen, 2013), and (Skibiński and Swacha, 2007). V(N) denotes the value V*10^N, Size is the size of file in Gigabytes, E:A denotes the number of elements and attributes, CD denotes the content density, the amount of character data contained in the XML, MD denotes the markup density, the amount of data required for structuring the XML (e.g., the amount of syntactic data and data used to name tags and attributes), and SP denotes the number of similar paths.

Table 1: Overview of XML Test Suite.

XML File	Size	E:A	CD	MD	SP
planet(1)	466.04	6.13(9) : 2.4(10)	0.507	0.493	63
uniprot(2)	155.54	3.15(9) : 4.17(9)	0.502	0.498	149
enwiki-latest(3)	44.19	2.35(8) : 2.08(7)	0.918	0.082	39
dblp(4)	1.28	3.25(7) : 8.08(6)	0.566	0.434	204
1gig(5)	1.09	1.6(7):3.83(6)	0.74	0.26	548
Swissprot(6)	0.106	2.97(6) : 2.18(6)	0.444	0.556	264
lineitem(6)	0.03	1.02(6):1	0.19	0.81	19

4.2 Results of Testing and Analysis

4.2.1 Compression Ratio

This section provides an experiment to test the compression ratio for the complete test XML files and

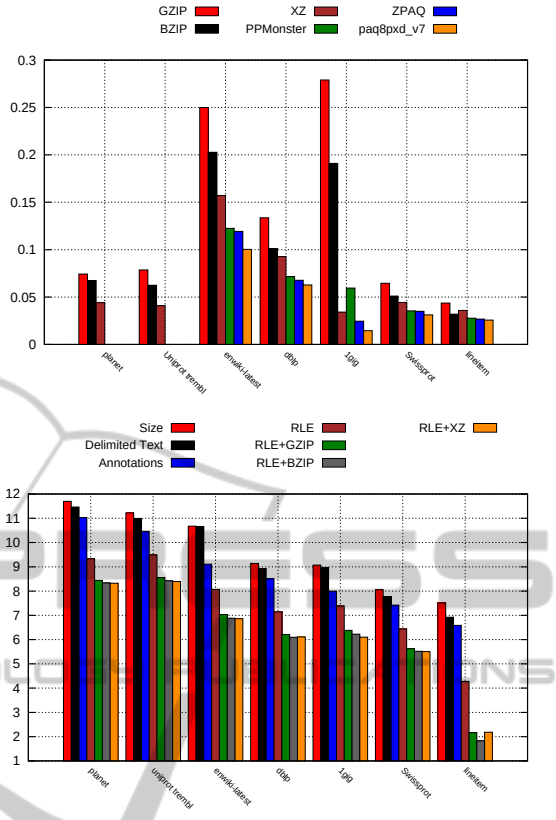

Figure 3: Top: Total compression ratios. Bottom: Log 10 byte sizes of individual components.

for various components of these files. In comparing XSAQCT and column-stores, the annotated tree is the only element independent of a column-store, therefore we need to measure how much memory is required to represent the XML data. For this experiment, the annotations will simply be stored as a sequential-list without any additional paged index scheme. For completeness, we also show how compressible the data is. Table 2 provides a breakdown of the largest annotation list (A), and its RLE-compressed equivalent (B). (Z) provides the annotation list that compresses the worst (least redundant annotations), and (Y) is its associated uncompressed size. This table shows that the performance of queries on the annotated tree is dependent on the paths being queried, especially when a RLE, which pre-computes much of the summation, is used.

The top and the bottom of Figure 3 respectively plot the compression ratio of each file using a standard XSAQCT process (with varying back-end compressors) and the total size of the annotation list, after applying a run length encoding to each annotation list, and then compressing each RLE encoding using a heavier weight compression scheme. For timing reasons, the largest files were not compressed with the

Table 2: Compression Ratio (all units in bytes).

File	A	B	Y	Z
planet	1.05(10)	5.18(5)	8.80(6)	6.85(6)
uniprot	3.03(9)	1.48(5)	1.48(7)	7.32(6)
enwiki-latest	5.72(5)	2.8(3)	5.72(5)	3.10(5)
dblp	5.60(4)	27745	5.56(4)	5.28(4)
1gig	2.39(4)8	1.17(2)	1.02(4)	6.40(3)
SwissProt	6(5)	2.95(1)	3.67(3)	3.81(3)
linteitem	2.4(3)	1.18(1)	2.40(3)	1.18(1)

more complex compressors (such as ZPAQ, paq8, PPMonster). With the more content-dense XML files, XSAQCT and vanilla compressors often compress the data to similar sizes, mostly because they are both equally limited in the compressing of free-formed English. However, when markup becomes equally dense, XSAQCT can compress much better, in addition to already allowing random access queries. Uniprot_trembl, for example, has a markup density of roughly 77 Gigabytes, its annotated representation requires approximately 26.75 Gigabytes, and a run-length encoding of each individual annotation list produces an annotated representation of roughly 2.9 Gigabytes, a markup compression ratio of 3.8 percent. One more GZIP compression on the RLE, produces an annotated representation of roughly 345 Megabytes, a markup compression of ratio of 0.5 percent. Finally, compressing all of its contents (text included) with GZIP produces a compression ratio of slightly 8 percent, or 12.2 GB.

Table 2 provides a breakdown of the largest annotation list (A), and its RLE-compressed equivalent (B). (Z) provides the annotation list that compresses the worst (least redundant annotations), and (Y) is its associated uncompressed size.

4.2.2 Query Efficiency

While efficiency of queries has been demonstrated in the earlier work, see (Fry, 2011) and (Müldner et al., 2010), this section provides a detailed of efficiency of queries in the context of our DBMSs.

The performance of query execution depends on two factors: (A) how fast the query can be resolved within the annotated tree; and (B) how fast the required contents can be extracted from each storage container (e.g., the text at location X, all text contents between two values).

Since (A) is based solely on

recursive summations and the number of subtrees that can be resolved at once (i.e., the number of times the condition on Line 6 in Algorithm 2 is satisfied), (A) is indeed efficient, since in one clock second, the summation (or product, assuming RLE) of more than 300 million *random* integers can be computed. However, Table 2 shows that for even the largest uncompressed annotation list (9.86 GB in total or 2,646,326,444 integers), if it were entirely resident in RAM, it would still require roughly nine seconds to compute its entire sum. In addition, the underlying column-stores and annotated tree still require memory for additional indexes, caches, and pending lists and although this will not require a substantial amount of memory, a scalable solution appears to be necessary. Sparse indexing, where an annotation-list is paged and each page is indexed based on its number of elements and total summation, is used for its simplicity and light memory footprint. This allows efficient random access, e.g., find the correct page and then finish the summation, and is inherently scalable (it is possible to index the index), all at the cost of additional I/O (bring in specific pages from disk and cache them).

(Fry, 2011) analyzed (B) in the compressed domain, and showed that when combined with a standard key-valued NOSQL database, selecting specific text contents often outperformed many of the popular XML Database engines. However, that analysis assumed that the entire annotated tree was RAM-resident; possibly an unfair assumption for very large XML files (although, indexing summations can be much more efficient than computing entire sums), and only tested selecting specific values. Note that selecting specific ranges of attributes with a column-store is often much more efficient than standard row-stores.

4.2.3 Storage

The central goal of a database system is processing queries as fast as possible, rather than achieving XSAQCT's goal, i.e, getting the highest compression ratio. A benefit of compressed data is that less time is spent on I/O operations during query processing, since less data is read from disk into memory. In addition, the growing discrepancy between CPU speed and memory bandwidth implies that more CPU cycles are being *wasted*, this implies that we have more CPU cycles to spare for decompression. Thus, for optimal query performance, compression algorithms that aim for direct access, (or if CLOBS are assumed) high decompression speed and a reasonable compression ratio are often preferred. For optimal extendibility, a DBMS that allows external compressor integration is one that is preferred.

Data Types. The easiest workaround to the problem of column stores that require a schema is by forcing every data-type to be a CLOB (which was done for the compression experiments above). Approximating the data-type, however, is ultimately a two-pass process (approximate the data type, then build the tables), which may be allowable given the domain we

are operating in, assuming the additional data type constraint is allowable. Any dynamic approach (e.g., each text element is further organized by perceived data-type), will be ignored since compression on CLOBs is still an effective solution.

Containers. For each node in an annotated tree, a new "table" (or storage structure) is created with its data-type specified according to the outline above. Each element in a text container is stored with its index as the primary key and in this case, the underlying storage and compression mechanisms are maintained by the database architecture. One detriment to this approach, however, are the overheads in explicitly representing each key, which can severely bloat the size of the data on disk. Many column-store architectures avoid storing this ID attribute by using the position (offset) of the tuple in the column as a virtual identifier. XSAQCT may also handle storage and compression, for instance, data is compressed and stored in tables as BLOBs (with a key associated with each). However, this approach may not allow for efficient querying upon the character data.

Annotation Lists. Compression, paging, indexing and storing the annotation lists can be implemented with-respect-to or independent-of the underlying column-store architecture. For example XSAQCT can: (1) store annotations individually in each table, using a sparse-index on the annotation sums and range queries to find the proper summations. Paging and caching is strictly defined by the column store used; (2) Store BLOBs of annotations in each table, and the indexes upon these BLOBS can be represented as *additional keys* (i.e., BLOB ID, Number Elements, Summation). Although with a traditional column store, this may create multiple columns, many modern column-store architectures allow *column-groups*, where multiple columns are stored on the same page, forming a row format of specific attributes, or (3) XSAQCT can natively handle (2), and the annotated tree defines the annotation indexing mechanisms, allowing the column-stores to still cache the BLOBs. One important consequence results from these actions: each annotated page brought into memory also implicitly defines the text elements that can be potentially queried upon. Integrating the caching mechanisms allows the column-store to potentially pre-fetch many text pages, which can improve performance especially when querying exhibits locality of reference. Additional performance tweaks include the relative weight in caching annotations, their indexes, and textual data.

5 CONCLUSIONS AND FUTURE WORK

This paper showed the relationships between XML compressors and column-stores. We showed that a permuting XML compressor, called XSAQCT with the DBMS back-end has essentially the same functionality as a column-store (while ignoring things such as SQL Joins), including a specific kind of compression, i.e., using similar path-based compression that resembles column-based compression in column-stores. To test the compression ratio achieved with this compressor, experiments were performed on an XML corpus. The annotated trees of our corpus range from 99.8 GB to 3.7 MB, which compressed using RLE produces data ranging from 2.88 GB to 18 KB. Compressing the RLE's with a heavier weight compression algorithm produces data ranging from 253.7 MB to 67 Bytes, showing that indexing exceptionally large XML files can be done in a succinct fashion. In addition, the system supports random access queries and the speed of query resolution within the annotated tree (founded on simple summations which are inherently simple operations). Regarding efficiency of queries, (Fry, 2011) showed that selection queries and update queries (using pending lists), outperformed many of the popular XML Database engines, creating an area of research for compressed XML in the database domain; see also (Müldner et al., 2010).

Many interesting problems remain for the future work, especially in the areas of joins. For example, the query: `/cars/car[data/attrib='Model' and data/text='855']` can be resolved by both a row-oriented and column-stores. A late-materialized column-store would first search the attribute column and store all public keys (or virtual IDs) satisfying the equivalence, and then it would scan the text column and remove from the list all keys that do not satisfy that equivalence. With respect to schema-less XML and the recursive sums property, the index of a text element in one text container is not necessarily equal to the index in another container for the same subtree (they are both related by the summation of their parents annotation list and the full-mixed content assumption), thus it is harder to properly execute a join.

The proposed approach could also be useful for Substantial Updates, Rollbacks/Checkpoints and Views. Finally, in addition to restricting nodes and subtrees for some specific user group, since an annotated tree is described in terms of a recursive relationship among sums, and summations of integers are decomposable and re-orderable, annotated trees are decomposable into views (with additional mechanisms to allow the annotated to be recombined).

ACKNOWLEDGEMENTS

The work of the first and second authors are partially supported by the NSERC CSG-M (Canada Graduate Scholarship-Masters) and NSERC RGPIN grant respectively.

REFERENCES

(2014). eXist DBMS, retrieved March 2014 from http://exist-db.org/exist/apps/homepage/index.html.

Abadi, D., Boncz, P. A., Harizopoulos, S., Idreos, S., and Madden, S. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280.

Amazon (2014). Amazon Redshift, retrieved March 2014 from http://aws.amazon.com/redshift/.

Choi, B. and Buneman, P. (2003). XML Vectorization: a Column-Based XML Storage Model. Technical report, University of Pennsylvania, Department of Computer & Information Science.

Corbin, T., Müldner, T., and Miziołek, J. (2013). Parallelization of Permuting Schema-less XML Compressors. In *PPAM 13*.

Florescu, D. and Kossmann, D. (1999). Storing and Querying XML Data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 22(3):27–34.

Fry, C. (2011). Extending The XML Compressor Exact With Lazy Updates. Master's thesis, Acadia University, Canada.

Hartmut, L. and Suciu, D. (2000). XMill: an efficient compressor for XML data. *ACM Special Interest Group on Management of Data (SIGMOD) Record*, 29(2):153–164.

Müldner, T., Fry, C., Miziołek, J., and Corbin, T. (2010). Updates of compressed dynamic XML documents. In *The Eighth International Network Conference (INC2010)*, Heidelberg, Germany.

Müldner, T., Fry, C., Miziołek, J., and Corbin, T. (2012). Parallelization of an XML Data Compressor on Multicores. In Wyrzykowski, R., Dongarra, J., Karczewski, K., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, volume 7204 of *Lecture Notes in Computer Science*, pages pp. 101–110. Springer Berlin Heidelberg.

Müldner, T., Fry, C., Miziołek, J., and Durno, S. (2009). XSAQCT: XML queryable compressor. In *Balisage: The Markup Conference 2009*, Montreal, Canada.

Müldner, T., Miziołek, J., and Corbin, T. (2014). Annotated Trees and their Applications to XML Compression. In *The Tenth International Conference on Web Information Systems and Technologies*, Barcelona, Spain. WEBIST.

OSM (2014). OpenStreetMap Foundation, retrieved March 2014 from http://wiki.osmfoundation.org/wiki/Main_Page.

Raman, V., Attaluri, G. K., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., Malkemus, T., Müller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A. J., and Zhang, L. (2013). Db2 with blu acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091.

Skibiński, P. and Swacha, J. (2007). Combining efficient XML compression with query processing. pages 330–342.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E. J., O'Neil, P. E., Rasin, A., Tran, N., and Zdonik, S. B. (2005). C-store: A column-oriented dbms. In Bøhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P. A., and Ooi, B. C., editors, *VLDB*, pages 553–564. ACM.

The DBLP Computer Science Bibliography (2014). The DBLP Computer Science Bibliography, retrieved March 2014 from http://dblp.uni-trier.de/db/.

The UniProt Consortium (2013). Update on activities at the Universal Protein Resource (UniProt) in 2013. http://dx.doi.org/10.1093/nar/gks1068. Retrieved on June 20, 2013.

Wikipedia: The Free Encyclopedia (2014). The Free Encyclopedia, retrieved March 2014 from http://en.wikipedia.org/wiki/Main_Page.

XML (2013). Extensible markup language (XML) 1.0 (Fifth edition), retrieved October 2013 from http://www.w3.org/TR/REC-xml/.

XML Schema (2013). XML Schema, retrieved October 2013 from http://www.w3.org/XML/Schema.

xmlgen (2013). The benchmark data generator, retrieved October 2013 from http://www.xml-benchmark.org/generator.html.

XPath (2013). XML Path Language (XPath), Retrieved October 2013 from http://www.w3.org/TR/xpath/.

XQuery (2013). XQuery 1.0: An XML Query Language (Second Edition), Retrieved October 2013 from http://www.w3.org/TR/xquery/.

XQuery Update (2013). Xquery update facility 1.0, retrieved October 2013 from http://www.w3.org/TR/xquery-update-10/.

Zukowski, M., van der Wiel, M., and Boncz, P. (2012). Vectorwise: a Vectorized Analytical DBMS, retrieved March 2014 from http://www.w3.org/TR/REC-xml/.