XXS: Efficient XPath Evaluation on Compressed XML Documents

NIEVES R. BRISABOA
ANA CERDEIRA-PENA
University of A Coruña, Spain
and
GONZALO NAVARRO
University of Chile, Chile

The *eXtensible Markup Language* (XML) is acknowledged as the *de facto* standard for semi-structured data representation and data exchange on the Web and many other scenarios. A well-known shortcoming of XML is its verbosity, which increases manipulation, transmission, and processing costs. Various structure-blind and structure-conscious compression techniques can be applied to XML, and some are even access-friendly, meaning that the documents can be efficiently accessed in compressed form. Direct access is necessary to implement the query languages XPath and XQuery, which are the standard ones to exploit the expressiveness of XML. While a good deal of theoretical and practical proposals exist to solve XPath/XQuery operations on XML, only a few ones are well integrated with a compression format that supports the required access operations on the XML data. In this work we go one step further and design a compression format for XML collections that *boosts* the performance of XPath queries on the data. This is done by designing compressed representations of the XML data that support some complex operations apart from just accessing the data, and those are exploited to solve key components of the XPath queries. Our system, called XXS, is aimed at XML collections containing natural language text, which are compressed to within 35%-50% of their original size while supporting a large subset of XPath operations in time competitive with, and many times outperforming, the best state-of-the-art systems that work on uncompressed representations.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.3 [**Database Management**]: Systems—*Query processing*; H.3.2 [**Information Storage and Retrieval**]: Information storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.3.7 [**Digital Libraries**]: Collection

General Terms: Algorithms, Performance

Additional Key Words and Phrases: semi-structured data, XML, XPath, compression, self-index

ACM Reference Format:
Nieves R. Brisaboa, Ana Cerdeira-Pena, and Gonzalo Navarro, 2013. XXS: Efficient XPath Evaluation on Compressed XML Documents. ACM Trans. Inf. Syst. V, N, Article A (January YYYY), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

A preliminary partial version of the data structure we use appeared in *Proc. ECDL'09* [Brisaboa et al. 2009].
Authors' addresses: Nieves R. Brisaboa, Ana Cerdeira-Pena, Department of Computer Science, University of A Coruña, Facultade de Informática, Campus de Elviña, s/n 15071 A Coruña, Spain. {acerdeira, brisaboa}@udc.es. Gonzalo Navarro, Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The *eXtensible Markup Language* (XML) [W3C 1998] has become a *de facto* standard to represent semi-structured information, due to its flexibility and suitability for data representation and communication between applications and services across different platforms. As a result, increasingly larger XML document databases are stored, transmitted, and manipulated in a wide range of applications.

To exploit the expressive power of XML, powerful query languages like XPath [W3C 1999] and XQuery [W3C 2010b] have been designed to allow constraint formulation on both document content and structure. Their growing interest and the challenge of supporting those query languages have triggered much research during the last years aimed to provide efficient solutions, either as theoretical works or practical systems.

Systems implementing XPath/XQuery have been usually divided into two categories (Table I illustrates some representative solutions from each one): those that follow a *streaming* approach, and thus sequentially read the documents to answer each query, and the *indexed* ones, which first preprocess the documents to build additional data structures over it; these are later used to solve queries without sequentially traversing the whole collection. Although streaming systems use little main memory, their processing times are conditioned by the need to sequentially scan the data. In turn, indexed systems may improve query times, but at the expense of increasing space requirements. Yet, note that in case the space needed for the index makes it necessary to manipulate it on disk, the efficiency of indexed approaches could be seriously degraded by I/O transfer times.

Reducing the space usage of XML data and additional structures is crucial to fit the indexes in memory rather than swapping out to disk, thus operating in higher and faster levels of the memory hierarchy, using fewer machines in distributed scenarios, or even to achieve a feasible solution at all when the memory is limited (as in mobile devices). In addition, working with a compressed version of a document saves time when it is transmitted through a network, when we need to access to disk looking for a document, and more importantly, when it is processed.

A common classification of the XML compression tools regards the awareness of its structure, which leads to the distinction between *XML blind* (e.g., Ziv-Lempel techniques [Ziv and Lempel 1977; 1978; Welch 1984], Huffman compression [Huffman 1952; de Moura et al. 2000], PPM based methods [Cleary and Witten 1984], Dense Codes compression [Brisaboa et al. 2007], etc.), and *XML conscious* compressors (some of the most relevant are shown in Table II). The last ones are subject to a further division. Given the relevance of the XML query languages, most of the XML conscious compressors have gone one step beyond, and provide some query support rather than just reducing space. These tools are known as *queriable* compressors, in contrast with *non-queriable* compressors. Some of them allow one to perform queries directly over the compressed representation of the text (either sequentially or using indexes), while others need to decompress the data (either fully or partially) before operating over them[1]. However, despite the large amount of research developed on these issues, today there is a stated lack of available practical solutions [Sakr 2009].

A recent research line explores representations combining compression and indexing, by creating so-called *self-indexes* [Navarro and Mäkinen 2007]. A self-index is a compressed representation of the data that can be searched with efficiency comparable to that of an indexed representation. Thus it can be regarded as a compressor that speeds up querying, instead of slowing it down, or as an index that reduces the space usage, instead of increasing it.

[1]A complete review can be found in Cerdeira-Pena [2013].

Table I. State-of-the-art solutions for XML storage and querying: General systems

	Streaming	Indexed
Systems	**SPEX** [Olteanu 2007], **XSQ** [Peng and Chawathe 2005], **GCX** [Schmidt et al. 2007]	**Saxon** [Kay 2008], **Galax** [Fernández et al. 2003] **eXist** [Meier 2002], **Qizx/DB** [XML Mind products 2008], **MonetDB/XQuery** [Boncz et al. 2006]

Table II. State-of-the-art solutions for XML storage and querying: XML conscious compressors

	Non-queriable	Queriable
XML Compr.	**Millau** [Girardot and Sundaresan 2000], **XMill** [Liefke and Suciu 2000], **XMLPPM** [Cheney 2001], **Exalt** [Toman 2003], **XWRT** [Skibinski et al. 2008], **SCA** [Levene and Wood 2002], **SCM** [Adiego et al. 2007b], **AXECHOP** [Leighton et al. 2005], **XComp** [Li 2003], **RNGzip** [League and Eng 2007] **XAUST** [Subramanian and Shankar 2005]	**XGrind** [Tolani and Haritsa 2002], **XSeq** [Lin et al. 2005], **XPRESS** [Min et al. 2003], **XQueC** [Arion et al. 2007], **XQzip** [Cheng and Ng 2004], **ISX** [Wong et al. 2007] **QXT** [Skibinski et al. 2008], **XCQ** [Ng et al. 2006], **XCPaqs** [Wang et al. 2004], **LZCS** [Adiego et al. 2007a], **TREECHOP** [Leighton et al. 2005] **XBzipIndex** [Ferragina et al. 2006; 2009] **TinyT** [Maneth and Sebastian 2010] **SXSI** [Arroyuelo et al. 2010]

A self-index for XML represents both the structured text and an index built over it. The first such self-index [Ferragina et al. 2006; 2009], known as XBzipIndex, provides direct access and support for a very limited class of XPath queries. Arroyuelo et al. [2010] proposed another self-index for XML data. This tool, called SXSI, is tailored to work in main memory and addresses an important subset of XPath. Its main drawback is that its space usage is still high compared to the size obtained by a plain compressor. Later, Maneth and Sebastian [2010] presented TinyT, a structural self-index for XML based on grammar-based tree compression, optimized to specifically handle some structural XPath queries.

When considering the overall picture, one can observe that efficient, scalable and stable implementations taking little space and simultaneously providing a comprehensive XML query support, are highly desirable, although they have not been achieved yet. Regarding general systems, streaming solutions suffer from prohibitive processing times, while the weakness of indexed proposals arises from their high space consumption. XML compression solutions provide limited or no query support, and many systems are not actually available. *Self-indexed* representations are promising alternatives, but they are still far from offering a competitive and complete solution.

In this paper we introduce a system, dubbed *XXS: Efficient **X**Path Evaluation on **X**ML documents using a **S**elf-Index*, for the efficient evaluation of XPath queries within the space of the compressed collection (35%-50% of the original data size). XXS is aimed at working in main memory, is static (i.e., in case the XML data changes, it must be rebuilt) and focuses on XML collections of natural language text, which comprises a significant fraction of the available XML data[2]. This means that XXS indexes meaningfully only XML collections where the text nodes contain natural language, and that only whole-word (and phrase) queries are supported on the text contents.

The experimental evaluation proves that XXS has an outstanding performance. It successfully competes with well-known state-of-the-art solutions (MonetDB/XQuery, Qizx/DB and SXSI), which XXS outperforms by far in terms of space requirements, using 2-5 times less space. Our experiments have focused on a wide fragment of XPath, including a practical subset of Core XPath [Gottlob et al. 2005] and some additional

[2]See, for example, a list of large document-centric (or text-centric) XML databases by Bourret [2009, Section 3.1]; data-centric applications are listed in Section 4.1. On the other hand, Oszu [2003, Section 4.2.1] mentions that public testbeds are much more easily available for text-centric than for data-centric XML.

functions such as `equal`, `contains`, and `count`. However, we remark that XXS works over an exact representation of the text, thus allowing any other query extension to be developed as well. In the Conclusions we briefly discuss some future lines of our work.

This paper is organized as follows. The next section introduces some preliminary notions about data structures and essential concepts for the development of our tool. Sections 3 to 5 are devoted to explain the XXS tool, with a detailed description of its main components. Section 6 presents the results of our experiments. We conclude in Section 7 and provide future research directions. A glossary with the most frequeny acronyms used is given in the Appendix.

2. PREVIOUS CONCEPTS

2.1. Dense Code Word-based Bytewise Encoders

Word-based bytewise coding methods use words as source symbols, while codes are sequences of bytes. For natural language text, it has been shown that using words[3], instead of characters, significantly improves compression ratios, as words exhibit a more biased distribution of frequencies [Baeza-Yates and Ribeiro-Neto 1999]. On the other hand, decompression and searching can be boosted by using byte-oriented encoders, since no bit manipulations are needed. Some of the most representative bytewise word-based encoders are Huffman-based codes (in particular, *Plain Huffman* and *Tagged Huffman*) [Huffman 1952; de Moura et al. 2000], Dense Codes [Brisaboa et al. 2007], and Restricted Prefix Byte Codes [Culpepper and Moffat 2005].

The Dense Codes family is especially convenient for our work, given the distinction of bytes that it considers: *stoppers*, or bytes that only can appear at the end of a codeword, and *continuers*, or bytes that cannot end a codeword. *End-Tagged Dense Code* (ETDC) [Brisaboa et al. 2007] is the simplest member of the Dense Codes family. It reserves the same amount of byte values to be used as stoppers (values from 0 to 127) and as continuers (values from 128 to 255). However, this proportion between stoppers and continuers could not be optimal for a given word frequency distribution of the text. The *(s,c)-Dense Code* (SCDC) [Brisaboa et al. 2007] is a generalization of ETDC where digits between 0 and $s-1$ are used as stoppers and digits between s and $s+c-1 = 255$, are used as continuers. The pair (s, c) (where $s + c = 256$) is chosen so as to optimize compression ratios.

For semi-static statistical compression, the encoding process of both ETDC and SCDC performs a first pass over the source text to gather the different words and their frequencies (the *model*). The frequencies are used to sort the vocabulary and then a codeword is assigned to each word (shorter codewords to more frequent words). The codeword assignment is performed sequentially, thus making the computation very simple. For instance, if we consider the SCDC technique, the first s words in the vocabulary are given one-byte codewords, from 0 to $s-1$. Words ranked from s to $s+sc-1$ are sequentially assigned two-byte codewords. The first byte of each codeword is a value in the range $[s, s + c - 1]$, that is, a continuer. The second byte, the stopper, is a value that belongs to $[0, s-1]$. The next words are encoded with three-byte codewords, and so on. After this process, a second pass is performed where the compressor replaces each word by its codeword, yielding the compressed representation of the text.

2.2. Wavelet Trees on ByteCodes

The *Wavelet Tree on Bytecodes* (WTBC) [Brisaboa et al. 2012] reorganizes the codeword bytes of a text compressed with any word-based byte-oriented technique. This

[3]We speak of words to simplify the discussion. In practice both words and separators are encoded as atomic entities in word-based compression.

codeword rearrangement basically consists of placing the different bytes of each codeword at different nodes, following a wavelet-tree-like [Grossi et al. 2003] structure, instead of sequentially concatenating them, as in a typical compressed text. The reorganization turns to offer implicit indexing properties, so that random access to any word of the text is supported, and search times are drastically improved, by using a negligible amount of additional space. Brisaboa et al. [2012] showed that WTBC not only performs much more efficiently than sequential searches over compressed text, but also than explicit inverted indexes when little extra space is used. WTBC especially succeeds when searching for single words and short phrases.

The essence of this codeword rearrangement is the following: the root of the WTBC is an array containing the first bytes of the codewords, in the same order as the words they encode in the original text. That is, let us assume we have the text words $\langle w_1, w_2 \ldots w_n \rangle$, whose codewords are $cw_1, cw_2 \ldots cw_n$, respectively, and let us denote the bytes of a codeword cw_i as $\langle cw_i^1 ... cw_i^m \rangle$ where m is the size of the codeword cw_i in bytes. Then the root is formed by the sequence of bytes $\langle cw_1^1, cw_2^1, cw_3^1 ... cw_n^1 \rangle$. At position i, we place the first byte of the codeword that encodes the ith word in the source text, so notice that the root node has as many bytes as words has the text.

We consider the root of the tree as the first level. The second bytes of the codewords longer than one byte are placed in the nodes of a second level. The root has as many children as different bytes can be the first byte of a codeword of two or more bytes. For instance, in a $(192, 64)$-DC encoding scheme, the root will have always 64 children, because there are 64 bytes that are continuers. Each node X in this second level contains all the second bytes of the codewords whose first byte is X, following again the same order of the source. That is, the second byte corresponding to the jth occurrence of byte x in the root, is placed at position j in node X. That is, assume there are f words coded by codewords $cw_{i_1} ... cw_{i_f}$ (longer than one byte) whose first byte is x. Then, the second bytes of those codewords, $\langle cw_{i_1}^2, cw_{i_2}^2, cw_{i_3}^2 ... cw_{i_f}^2 \rangle$, form the node X in the second level. The same idea is used to create the lower levels of the tree. Assuming there are d words whose first byte codewords is x and whose second one is y, then node XY is a node of the third level, child of node X, and it stores the byte sequence $\langle cw_{j_1}^3, cw_{j_2}^3, cw_{j_3}^3 ... cw_{j_d}^3 \rangle$ given by all the third bytes of these codewords. Those bytes are again in the original text order. Therefore, the resulting tree has as many levels as bytes have the longest codewords. Figure 1 shows an example of a WTBC[4] built from the text '`MAKE EVERYTHING AS SIMPLE AS POSSIBLE BUT NOT SIMPLER`'.

2.2.1. WTBC Basic Procedures. The two main operations using a WTBC are *decoding* the word placed at a given position of the text, and *locating* the occurrences of a word. Both algorithms are based on the use of *rank* and *select* operations over the node byte sequences, respectively. Given a byte sequence $B = \langle b_1, \ldots, b_n \rangle$:

— $rank_b(B, i)$ = number of occurrences of byte b in B up to position i.
— $select_b(B, j)$ = position of the j^{th} occurrence of the byte b in byte sequence B.

The efficiency of the WBTC hinges on the implementation of *rank* and *select* operations. A two-level directory of partial counters is maintained for each byte sequence in order to avoid the sequential counting of the number of occurrences of a searched byte from the beginning of a WTBC node[5]. There is a tradeoff between space and time. The more the partial counters, the more space is needed, but *rank* and *select* operations will be more efficient.

[4]Notice that only the shaded byte sequences are stored; the rest of the text is shown for clarity.
[5]See Brisaboa et al. [2012] for implementation details.

In order to decode a word we go down in the tree by using *rank* operations. For instance, to know which is the 7^{th} word in the example of Figure 1, we start by reading the byte at that position in the root node. That is, $root[7] = b_3$. According to the encoding scheme[6], we know that the codeword is not complete yet, so we will move to the second level of the tree, more precisely, to node $B3$. This node contains the second bytes of all the codewords whose first byte is b_3, following the order of the text. Thus, to find out which position of that sequence we have to read, we use $rank_{b_3}(root, 7) = 2$. In this way, $B3[2] = b_4$ gives us the second byte of the codeword we are decoding. Again b_4 is a continuer, so we proceed in a same way, but in the node $B3B4$, which corresponds to the first two bytes of the codeword we have just read (b_3b_4). There we read the byte that is at position $rank_{b_4}(B3, 2) = 1$, that is, $B3B4[1] = b_2$. Byte b_2 marks the end of the searched codeword. As a result, we finally obtain the codeword $b_3b_4b_2$, corresponding to `'BUT'`, which is precisely the 7^{th} word in the source text, as expected.

There are special procedures to perform full-text extraction and decompression of a large contiguous area. These take advantage of the fact that the byte sequences of the WTBC nodes follow the original order of the words in the source text, and are efficiently implemented using pointers to the next positions to be read in each node.

For locating the occurrences of a word we traverse the tree upwards, by means of *select* operations. For example, assume we want to find the first occurrence of the word `'SIMPLER'`. In Figure 1, we can observe that its codeword is $b_4b_5b_1$, so we start the search at node $B4B5$, where we locate the first occurrence of b_1 by computing $select_{b_1}(B4B5, 1) = 1$. Hence, the first position at node $B4B5$ corresponds to the first occurrence of `'SIMPLER'`. Next, we need to find the position of the first occurrence of byte b_5 in node $B4$, which is $select_{b_5}(B4, 1) = 3$. This indicates that our codeword is the third one starting by b_4 in the root node. We then proceed by locating the position of the third b_4 in the root of the tree, $select_{b_4}(root, 3) = 9$. Finally, we can answer that the first occurrence of `'SIMPLER'` is at the 9^{th} position in the source text.

Apart from decoding and searching, another basic procedure efficiently supported by the WTBC is to *count* the number of occurrences of a word. It just consists of counting how many times the last byte of the codeword assigned to the word appears in the corresponding WTBC node, using a simple *rank* operation on that byte sequence. Moreover, we can also count the number of occurrences of a word until a given position of the text. In that case, the same strategy is performed, but for each codeword byte, tracking down the endpoint toward the leaf node of the word.

Phrase pattern searches are also supported. A *phrase search* starts by locating each occurrence of the least frequent word of the phrase, and then checking in the wavelet tree root that the first bytes of the other words match. If they do, the rest of their bytes are verified downwards in the tree.

2.3. Succinct Tree Representations

Given the tree structure of XML documents, succinct tree representations are a key for the scope of this work. The classical representation of a general tree of n nodes uses $O(n)$ pointers (or words), each one requiring $w \geq \log n$ bits (our logarithms are base 2), thus leading to $O(nw)$ bits of space. The associated constant is at least 2, which permits to support basic operations such as moving to the first child and to the next sibling, or to the ith child. Some other simple operations (e.g., moving to the parent, obtaining the depth, etc.) and sophisticated ones (e.g., moving to a specific level-ancestor or to the lowest common ancestor of two nodes), are also supported, but by further increasing this constant. Since Jacobson [1989], much research has focused

[6]We assume that bytes b_1 and b_2 are stoppers, while bytes b_3, b_4, and b_5 are continuers (not all the combinations are used).

on reducing the space to represent trees, achieving $2n + o(n)$ bits of space and constant time for most of the operations. The distinct proposals mainly differ in the functionality provided and also in the nature of the $o(n)$ space overhead. In this work we consider the family of *balanced parentheses (BP)* representations [Jacobson 1989; Munro and Raman 2001; Sadakane and Navarro 2010]. This is built from a depth-first preorder traversal, writing a '(' when arriving to a node, and a ')' when we leave it. In this way, each node is represented by a pair of matching opening and closing parentheses, leading to a sequence of $2n$ balanced parentheses. Tree operations are solved by using some core parenthesis operations, namely *findopen*, *findclose*, and *enclose*. Early works [Munro and Raman 2001] achieved constant time support for basic tree operations (e.g., *parent*, *subtreesize*, *nextsibling*, etc.). Recently, a new proposal [Sadakane and Navarro 2010], called *fully-functional succinct tree*, was able to solve in constant time many other sophisticated operations (such as *child*, *lowest common ancestor*, or even *level ancestor*) that are not usually handled by other BP representations.

3. XXS: XML WAVELET TREE

The XXS tool provides a compact representation of XML documents, with an efficient query support. Two main parts compose our solution:

— *XML representation*: XML documents are represented in a compressed and self-indexed way by using a new data structure that we call XML Wavelet Tree (XWT). This data structure has been designed to support XML querying (Section 3).
— *Query module*: This part aims to efficiently solve XPath queries over an XWT representation. It is divided into two main components:
  — The *Query Parser* is in charge of the query parsing task, from the text representation of a query until the final query execution plan (Section 4).
  — The *Query Evaluator* is devoted to perform the actual evaluation task. The global evaluation procedure is characterized by three main strategies: a *bottom-up* approach, a *lazy evaluation* scheme, and a *skipping* strategy (Section 5).

This section deals with the first module, introducing the XML Wavelet Tree (XWT).

3.1. XWT Construction

The XWT data structure follows the essence of the WTBC reorganization of codewords explained in Section 2.2, using as compression method the SCDC compressor discussed in Section 2.1. As a result, the process of obtaining the final XWT representation of an XML document[7] is made in two phases. However, this data structure has been specifically designed to deal with XML documents and to efficiently support XML retrieval, by particularly focusing on XPath queries. To this end, various features are considered throughout the general construction process.

3.1.1. Phase I: Document Parsing and Codeword Assignment.

Document parsing. The first step in the XWT construction consists of parsing the input XML document to gather the different words that will compose the vocabularies and to compute their frequency distributions. To this aim we use a variant of the spaceless word model [de Moura et al. 2000], where single spaces are not coded but implicitly assumed between two consecutive non-separator codes.

The parsing distinguishes different kinds of words depending on whether a word is[8]:

— A *start-tag* or an *end-tag*.

[7]Notice that a collection of documents can be regarded as a single document that integrates all of them.
[8]This division is in accordance with the XPath data model [W3C 1999].

— The name of an attribute.
— An attribute value.
— A word inside a *comment*.
— A word inside a *processing instruction*.
— A word of the XML document text content.

With this aim, the basic spaceless word model is slightly modified, since we also consider the following cases as single words, independently of whether alphanumeric and non-alphanumeric characters are mixed: (*i*) the group of characters formed by the left angle bracket, <, and the name of a start-tag markup (e.g., <name), (*ii*) the end-tag markup as a whole (e.g., </name>), (*iii*) the name of an attribute followed by the equal character (e.g., name=), and (*iv*) the reserved initial and final character groups defining a special markup, such as *comments* (<!-- and -->), *processing instructions* (<? and ?>), *CDATA sections* (<![CDATA[and]]>), and so on.

As a result, the same word will be assigned different codewords depending on the category it belongs to. For instance, if the word romance appears as text content (e.g., ...an epic romance...), but also as an attribute value (e.g., category="romance") and inside a comment (e.g., <!--...it was a romance...-->) it will be stored as three different entries in the vocabularies, one for each category, leading to three different codewords. Making this difference between the same words according to their role increases the vocabulary size, but it will yield more efficiency and flexibility for queries.

We also perform some minor *normalization* operations, such as to convert empty-element tags into their corresponding pair of start-end tags (e.g., <price/> becomes <price></price>), or to delete redundant spaces and spaces inside tags (e.g., <price > becomes <price>). Such normalizations are accepted in the XML standard.

Taking the aforementioned word division into account, four different vocabularies are created while parsing the XML document:

— The *content* vocabulary, which holds words from the text content category together with attribute value entries[9].
— The *tags* vocabulary, keeping the different start-tags and end-tags.
— The *attributes* vocabulary, which stores word entries corresponding to attribute names.
— The *nsearch*[10] vocabulary, holding words appearing inside processing instructions and comments.

We refer as *special* vocabularies those apart from the *content* vocabulary. Figure 2 shows the XWT representation built from an XML document sample, where the four different vocabularies are created.

Codeword assignment. To assign codewords, we use SCDC as the base compression technique. Recall that this compressor uses different bytes for continuers and for stoppers. Note that by reserving some continuers to be the first byte of the codewords assigned to words of the special vocabularies (one different continuer for each of the vocabularies), we can keep them located under specific branches of the XWT; that is, we can isolate them.

[9] We remark that, although attribute values and text content words share the alphabet, different word entries are stored in case of same words appearing in both categories, hence receiving different codewords. For example, in Figure 2, the word love appears as an attribute value, but also inside the text content of opinion tag. Thus, we keep two different entries inside the *content* vocabulary (see $\texttt{love}_{\texttt{att}}$ and $\texttt{love}_{\texttt{text}}$ entries).
[10] *Non-searchable* vocabulary.

Therefore, once the parsing has finished, we start by assigning a codeword to the words of the content vocabulary following an SCDC encoding scheme, but keeping aside as many continuers as special vocabularies we have. For instance, in the example of Figure 2, where a (3,5)-DC encoding scheme is used to encode content words, the first three continuers, namely bytes b_3, b_4 and b_5, are discarded. Notice that they are never used as first byte of any of the codewords assigned to words of the content vocabulary. In turn, these bytes will mark the starting byte of codewords corresponding to words of the special vocabularies. We used byte b_3 to mark start/end-tags, byte b_4 for attribute names and byte b_5 for comments and processing instructions (see the bytes shaded in the `CODE` column of the special vocabularies).

As stated, this particular feature allows the isolation of the special words, which has important benefits. In case of the tags vocabulary, for instance, one can observe that the subtree below $B3$ is devoted to exclusively store start-tags and end-tags. Remember that they follow the document order, and hence they maintain their relationships as in the original XML document. So, we can say that this subtree actually stores the complete XML document structure. The isolation of attributes, in turn, gives the flexibility to directly operate on them during query evaluation, while the isolation of comments and processing instructions provides a way to easily distinguish fragments that should be skipped in general text searches.

3.1.2. Phase II: Compression and XWT Creation. After the codeword assignment, we perform a second pass over the text replacing each word by its corresponding codeword and storing the codeword bytes along the different nodes of a tree, following the WTBC codeword bytes reorganization. The XWT nodes can be allocated and filled with the codeword bytes as the second pass takes place, since it is possible to precompute the number of nodes as well as their size in advance[11]. Therefore, by just keeping an array of markers indicating the next writing position for each node, they can be sequentially filled following the order of the words in the text.

As the XWT is based on the WTBC codewords reorganization, the basic procedures to *count*, *decode* and *locate* a word/phrase pattern that we can perform over the obtained representation are basically those described in Section 2.2.1. All of them have been extended to work over the XWT data structure.

3.2. Connection between XWT and a BP Representation

As previously pointed out, the subtree of the XWT that stores the document structure provides a structural isolation. What is more interesting is that the root of this subtree (node $B3$ in the example of Figure 2) matches a balanced parentheses (BP) representation of the XML document structure. That is, a position in that node exactly matches the same position in the BP stream. For instance, if we consider the BP representation of the document sample shown in Figure 2, ((((()())()))) (see Figure 3), we can observe that the third '(' is closed by the ')' at position 8, which precisely corresponds to `<author` start-tag, and `</author>` end-tag, respectively. Therefore both data structures can be used in combination to provide an efficient query support. We can perform basic tree operations over the BP (such as finding the *parent*, the *open*/*close* pair, or even the *depth* of a node), and then use the XWT to locate a position of the BP node into the original XML document, and to obtain the associated tag identifier.

For instance, let us consider the example of Figure 3. Assuming that we have just located the first occurrence of `<opinion`, we may be interested in the position of its corresponding end-tag, or the identifier of its parent. Note that `<opinion` is at position

[11]More precisely, just after the first phase has finished, as they are determined by both the encoding scheme and the frequencies of the words of the vocabularies.

9 in the structural node $B3$, but also in the BP. Therefore, in the first case, we can take advantage of the *findclose* operation provided by the BP representation and compute *findclose*$(9) = 10$, which tells us that the matching end-tag of `<opinion` corresponds to position 10 in $B3$. Once this position is known, we can easily obtain the position in the source text, by simply going one level up the XWT through a *select* operation. If, instead, we look for the parent of `<opinion`, we can use the *enclose* operation, which returns the position of the start-tag enclosing another one. So, *enclose*$(9) = 2$ gives us the location of the parent of `<opinion`, again in both the BP and $B3$. This information is then enough for our data structure to perform the *decode* basic procedure from that position of the structural node to finally discover the parent identifier of the target occurrence of `<opinion`, which is `<film`.

3.3. Segments in an XML Document

Another relevant feature to consider at this point is that any component of an XML document (e.g., an element, an attribute, a word, a phrase, etc.) can be ultimately regarded as a *segment* $[s, e]$, whose limits arise from the *start* (s) and *end* (e) positions in the text of the component. For instance, in case of an element, the positions of its corresponding start-tag and end-tag mark the limits of the segment that represents it (see segments depicted in pink on top of the XWT structure in Figure 3). In the same way, the segment representing a phrase pattern is determined by the positions of the first and last word of the pattern. Indeed, even when working with words, the same representation applies, since words are particular cases of segments starting and finishing at a single position.

Such a representation allows one to compare any two segments $a = [a.s, a.e]$ and $b = [b.s, b.e]$ by using the relations shown in Figure 4. In Section 5, this segment representation will become a key factor to perform query evaluation over the XWT.

4. XXS: QUERY PLAN CONSTRUCTION

The *Query Module* of the XXS system is devoted to evaluate XPath queries over the XWT. This module is composed by two main components: the *Query Parser* and the *Query Evaluator*. This section focuses on the *Query Parser* submodule, which covers the process from the initial query representation up to the construction of the final execution plan. Next we will conceptually explain the different phases of this process[12].

4.1. XPath Query Support

The XXS system supports a wide fragment of XPath, including the practical subset of the Core XPath defined by Gottlob et al. [2005] (with the exception of the *not* boolean operator). Therefore, we support all navigational axes, both element and attribute node tests, and filters with *and* and *or* boolean operators. Additional to Core XPath, we also support some of the most common text functions of XPath 1.0, namely the *equality* (`=`) and *contains* (`contains()`) functions, plus the *count* node set function (`count()`). Text functions can be applied over elements text content and also attribute values. In both cases, we assume word-based text searches (according to the XWT word-based model).

We show below the EBNF notation of the target fragment. As stated, `Axis` stands for any *forward* or *reverse* axis, `NodeTest` is either a tag/attribute name or the wildcard `'*'`, and `Pattern` can be any word or phrase pattern.

```
Core+            ::=  'count(' Core')' | Core
Core             ::=  LocationPath | '/' LocationPath
LocationPath     ::=  LocationStep('/' LocationStep)*
```

[12]We refer the reader to Cerdeira-Pena [2013] for a detailed revision.

```
LocationStep       ::=   Axis‘::’NodeTest |
                         Axis‘::’NodeTest‘[’Pred‘]’
Pred               ::=   Pred ‘and’ Pred | Pred ‘or’ Pred |
                         LocationPath | LocationPath‘=’Pattern |
                        ‘contains(’ LocationPath‘,’Pattern‘)’ |
                        ‘(’ Pred ‘)’
```

4.2. Initial Query Plan: The Query Parse Tree

XPath expressions are regarded as sequences of *location steps*, where the result of the current step makes up the context for the next one. Previous and current location steps are related by *axes*. Hence it is possible to get an initial representation of the query, which we call *query parse tree*, by taking the output of a query parser[13] and converting sequences of location steps into a composition of binary relations, whose operands are the corresponding *node tests* and the composition of the location path itself. That is, from left to right, the *query parse tree* is built upwards as follows. Each location step is translated into a main node labeled with the step axis name and two children. The left child represents the location step node test, whose occurrences are delivered by the axis node. In turn, the right child comes from the tree representation already set up from the previous location step. Figure 5 shows the query parse tree[14] corresponding to the query `/library/book[./data/following-sibling::summary]/descendant::title`. We remark that location paths inside predicates are similarly translated into a composition of relations as location paths outside predicates. This time, however, in order to allow their further integration within the global query parse tree, we must reverse both the order in which the location steps are considered to build the tree (now from right to left) and the meaning of the axes. Axes with opposite meaning are, for instance, *child* ↔ *parent*, *descendant* ↔ *ancestor*, *following* ↔ *preceding*, etc.

4.3. Query Plan Optimization: Query Parse Tree Transformations

The initial query parse tree of an input query can already be used as the *query execution tree* to be further evaluated. Nevertheless, we perform some transformations over it to gain efficiency during evaluation. Some of them are plain algebraic simplifications, while some other are transformations that modify the original query parse tree (since it only considers components of the XPath syntax), by producing an equivalent one in terms of retrieved results, but optimized to meet XWT features. In the process we introduce new operations that are not part of XPath but are efficiently solvable with the XWT. We have defined four main groups of transformations, including about 40 rules. A detailed description of each individual transformation can be found in Cerdeira-Pena [2013]. Next, we will briefly enumerate the distinct groups, and illustrate an example rule of each of them applied over the query sample depicted in Figure 6[15]:

(1) *Attributes equality simplification*: this converts an equality step between an attribute name and its value, such as `...[@city="Las Vegas"]/...` or `.../@*[.="Paris"]/...` into a phrase pattern search (Figure 6.1).
(2) *Wildcard optimizations*: we can distinguish the next three transformations over location steps involving wildcards (‘*’):

[13]To parse an input query into its different components we have used the source code provided by Benjamin Piwowarski, based on his `soul` library (`http://sourceforge.net/projects/soulparsing`).
[14]We refer as *root* the *root node* of an XML document, according to the XPath data model.
[15]For simplicity we use subscript “$_{att}$” to mark nodes representing attributes or operators (i.e., axes/functions) whose child nodes is ultimately an attribute.

(a) *Redundancy removal*: this optimization aims at discarding a costly (or unnecessary) step. For instance, given the fragment of the query parse tree depicted in Figure 6.2, we can avoid processing the `child` step over the wildcard (which potentially selects all element children from the root node to be further analyzed with respect to another wildcard element node), by combining it with the `descendant-or-self` axis into a single step, `descendant`.
(b) *Synonyms translation*: with this transformation we aim to replace an axis with an equivalent one (that is, delivering the same results), and to produce sequences of same steps that can be further optimized in *Steps unification*. Figure 6.3 shows an example of these equivalences.
(c) *Steps unification*: this optimization integrates several identical steps over the wildcard '*' into one, which is less costly[16]. For instance, let us consider the fragment marked on top of Figure 6.4. It retrieves all `summary` element nodes (having a `keyword` attribute with value "`XML`") at distance 3 descending from a valid `content` node. Instead of iteratively covering each `child` step involving wildcards, we can perform just one step, by creating a new operator, $\texttt{child}_{\texttt{dist}_3}$, which modifies the `child` semantics to also enforce a distance parameter[17]. The fragment highlighted at the bottom of Figure 6.4 shows another example of axis unification, this time regarding the `descendant` axis.

(3) *Or/and optimizations*: these include several transformations that simplify the query parse tree using properties of the logical `or` and `and` operators. Figure 6.5 depicts an example of transformation that applies over the `or` operator.
(4) *Root node deletion*: since the root node constitutes the root of the tree hierarchy of an XML document, any other element will descend from it. Hence, any location step involving a `descendant` selection from the root node can be removed.

4.4. Final Query Plan: The Query Execution Tree

Once the corresponding transformations over the query parse tree are performed, we obtain the query execution tree, or final execution plan (see Figure 6.6), which will become the input of the *Query Evaluator* submodule. At this stage, each node of the query execution tree is directly translated into an operator that stands for the specific component/axis/function it represents.

5. XXS: QUERY EVALUATION

The *Query Evaluator* component of XXS addresses the actual evaluation of the final query execution tree obtained from the *Query Parser* submodule. In this section we describe the global execution process, and discuss its most important features[18].

5.1. Conceptual Description

The query evaluation strategy used by XXS can be broadly regarded as a practical deployment of the general bottom-up evaluation strategy proposed by Gottlob et al. [2005]. They showed that naive implementations of XPath queries, via exhaustive enumeration of all the paths in the tree that match the query, lead to query times exponential on the query size q. Instead, they proposed a strategy that achieves $O(n^4q^2)$ time and $O(n^2q^2)$ space, where n is the number of nodes in the XML tree. The time complexity improves to $O(nq)$ in Core XPath and in an extension called XPatterns. Their

[16]Notice that '*' potentially selects all occurrences of any element/attribute, which makes a location step over it be extremely costly.
[17]We take advantage of the XWT ability to obtain the depth of an element/attribute, thanks to its linkage with a balanced parentheses representation.
[18]Again, we refer the reader to Cerdeira-Pena [2013] for further insights and specific details.

basic idea is to compute, for each node x of the query parse tree, a table called *context-value table*, which gives the answer of the query subtree rooted at x, for each possible context node in the XML tree. The table for each internal query tree node is built as a Cartesian product of the tables of its children nodes. Then they manage to avoid computing unnecessary entries from the tables, which are responsible for their high space requirements. Bottom-up strategies implemented in practical systems like Proximal Nodes [Navarro and Baeza-Yates 1997] avoid constructing the tables, but rather obtain their entries on the fly, ideally in left-to-right order (which is not always possible when axes pointing backwards in the XML document are present in the query).

XXS evaluation strongly relies on the Proximal Nodes model. This model builds on the following principles: (1) use of segments to define operations, (2) bottom-up evaluation, and (3) lazy evaluation. The use of segments was already explained in Section 3.3: any component of an XML document can be regarded as a segment $[s, e]$, given by the start (s) and end (e) positions of the text covered by the component. This representation is one of the key factors of XXS query evaluation. The other two principles define an operatory for query execution that is described next.

Given the query execution tree of an input query, the overall execution procedure starts by demanding the first result to the root node. This request is sent down through the tree nodes of the query execution tree until reaching the leaves. Note that tree nodes are either leaf nodes or internal nodes.

— **Leaf nodes**: they constitute the basic extraction operands. Each leaf node retrieves, from the XWT, the occurrences (segments) of the specific component that it represents, and delivers the valid segment found to the tree node above it.
— **Internal nodes**: these are operators that compare the segments they receive from both sides, using the comparison relations shown in Figure 4 (that is, $<$, $>$, $\subset$, $\supset$, and $=$). The semantics of the axis/function that the internal node represents indicates the type of relationship that received segments should satisfy. Yet, in some cases additional checks may also be needed (such as to have a given depth, or to share a common parent). Figure 7 illustrates the target relations that received segments must satisfy in order to meet the semantics of some of the most common XPath axes. If the segment comparison fulfills the required relationship, the internal node sends upwards the segment received from its *left* child[19]. Otherwise, the internal node will keep searching, consuming results from either child, until it finds a valid one. During this search, the decision of which side is asked for a new segment depends on the relationship between the current segments, and the relationship that they should satisfy to meet the node semantics.

With this operational scheme, results flow upwards until the root of the query execution tree finally delivers the first result. At this point, the whole procedure is repeated again searching for the next query result. We remark that results are retrieved one by one, leading to a lazy evaluation scheme, in which results are delivered on demand.

Example Let us consider Figure 8 to show this general behavior over the query `//image [contains (./parent::article, “Greek Islands”)]`. As stated, the execution procedure always starts by asking the root node of the query execution tree for the first result. Since it is an internal node, it must compare the segments received from both sides. Therefore, it first propagates the request downwards to obtain those segments. The left side of the root node is a leaf node, hence it retrieves the segment associated to the first occurrence of `image`, and delivers it to its parent (the root node, in this case). In

[19]The only exception is the or operator, which may deliver segments from both sides.

turn, the right side is an internal node again (the one labeled `contains`), so it proceeds by asking to its children the first `article` and "`Greek Islands`" segments, respectively, and then it compares them by checking whether the `article` segment contains the received segment of "`Greek Islands`". If it does, we have a hit, thus `contains` reports the `article` segment to the node above it, to continue the process in the same way up (see Figure 8.a). Otherwise, and depending on the comparison result, next occurrences of either child of `contains` will be requested, to proceed with comparisons until finding a valid `article` segment. For instance, in Figure 8.b we show the situation where $a.e < t.s$, therefore `contains` should ask for the next `article` occurrence to continue validations. Finally, when `contains` finds a valid `article`, the `child` node of the query execution tree can operate. In case that the received first segment of `image` is a child of the `article` segment delivered by `contains`, then we can produce the first query result (see Figure 8.c). Otherwise, (e.g., in Figure 8.d, `image` is a descendant of `article`, but not a direct child, as their depth difference is greater than 1) the process continues with the `child` node requesting the next `image` segment or `article` segment containing "`Greek Islands`", depending on the relation between the current segments.

5.2. Evaluation Strategies

The general evaluation scheme just described combines, as explained, a bottom-up approach, which starts from the leaf nodes of the query execution tree and works its way up to the root (see the flow of pink arrows in Figure 8), with a lazy evaluation plan, as results can be recovered by a loop that sequentially obtains them on demand. Yet, there is still another key factor that makes XXS so efficient. Recall that internal nodes keep on requesting segments from either child whenever current ones do not fulfill the desired relationship. These requests will be actually sped up by a *positional restriction* that the new retrieved segment must satisfy. This is our *skipping strategy*.

For instance, in Figure 9 we are interested in searching `book` elements that are an ancestor of an `award` node. Note that current segments (those marked in bold face), do not satisfy the `ancestor` axis condition, and that the `book` segment appears before the `award` one. So, we know that the `ancestor` node should request a new `book` segment. However, instead of just retrieving the next occurrence of `book` in a sequential order, it can proceed in a more efficient way. Observe that the second and third occurrences of `book` depicted in Figure 9 will not satisfy the `ancestor` semantics, as they finish before the end of `award`. Therefore, we can avoid visiting useless `book` segments, thus saving processing time, if the `ancestor` node seeks for the next occurrence of `book`, `b'`, finishing after the end limit of `award`, that is, fulfilling `b'.e > a.e`.

Formally, when a node of the query execution tree is required to deliver a new segment, it will perform a *position restricted retrieval* regarding the start or end position of the new requested segment, as applicable. We remark that, according to this evaluation model, segments are traversed in preorder, but only visiting relevant ones, that is, segments that we must necessarily visit in order to answer the query.

5.3. Implementation Details

The evaluation procedure can be ultimately regarded as a sequence of linked requests (see the flow of blue arrows in Figure 8) demanding new segments to either a leaf or an internal node, modified by positional restrictions that the retrieved segments must fulfill. These requests are actually implemented through a procedure we call *next*, whose most relevant details are analyzed next, by considering the operational scheme of both type of nodes.

5.3.1. Leaf Nodes. Leaf nodes are in charge of delivering the basic components, that is, elements, attribute names, words and phrase segments. Let us denote as $patt$ the specific component that the leaf node represents, and as p the positional restriction received. Then, the *next* procedure of a leaf node basically consists of:

(1) *Counting* the number of occurrences of $patt$ until p, that is $count(patt, p) = k$.
(2) *Locating* the $(k+1)^{th}$ occurrence of $patt$, that is $locate(patt, k+1)$.

Notice that both algorithms are efficiently provided by the XWT data structure. This general scheme applies for both words and attribute names. It also works in phrases, but focused on the least frequent word of the pattern, as in general searches of phrase patterns over the XWT. Yet, in this situation, we may also need to skip interleaved occurrences of start/end-tags, comments and processing instructions (e.g., in case of phrase patterns that may span more than one text node). Recall that we reserved specific first bytes to encode the words of those special vocabularies when we assigned codewords during the XWT construction. Therefore, the text fragments that we must omit now can be easily recognized while the first codeword bytes validation is performed in the root of the XWT. Observe that, by doing this, we still avoid further processing until the first bytes of the phrase codewords pass the test, following the same strategy as in a general search of a phrase pattern.

For elements, the received positional restrictions may be related to their start-tag or their end-tag, and both kinds may be inherited simultaneously from ancestors in the query execution tree. To achieve the best performance, when both restrictions are present procedure *next* will choose the one referring the most forward in the text, and will use the operations *findclose*/*findopen*[20] to find the other extreme of the segment and validate the other restriction as well.

A slightly modified procedure must be considered in case of *self-nested* elements. Under this scenario, the problem arises from a preorder delivery of the segments[21] when the search is performed with respect to the end-tag of an element that may contain occurrences of the same element inside it. The problem is that the general procedure would select first the most internal segment fulfilling the condition, instead of the next one in preorder. Thus, we need to check the ancestors of resulting segments to find occurrences of the same element that should be retrieved before (as they also satisfy the restriction, but appear before in a preorder traversal). Additionally we must store the inner (and subsequent) segments, to be delivered upon further requests.

5.3.2. Internal Nodes. The implementation of the *next* procedure on an internal node is more complex. Internal nodes may stand for any XPath axis, a function (e.g., `equal` and `contains`), and also any of the different new axes we create as a result of the query parse tree transformations (i.e., those modified with a *distance* parameter, as shown in Figure 8.4). Remember that internal nodes are basically operators that have to compare the segments received from both sides. In case those current segments do not satisfy the required relationship, internal nodes must determine which side will be asked for a new segment to continue the comparisons (according to the actual relationship between current segments and the one they should hold), and also the skipping positional restrictions. The generated positional restrictions will be different depending on each operator, but even for a same operator, we may find that these conditions are also different depending on whether it operates over elements that are self-nested or not. That is, in case of operators that retrieve element (tag) segments, or even which

[20] Again, thanks to the connection between the XWT structural node and the BP representation.
[21] According to XPath 1.0 [W3C 1999] results are node sets, hence with no order; while in XPath 2.0 [W3C 2010a], results are *sequences* of nodes in a particular order, the 'document order' (which applied over the XML document structure corresponds to a preorder traversal). Notwithstanding, arguably all the systems supporting XPath 1.0 assume as well this 'document order' for results delivery. We also assume that, even as a way to allow the compatibility of XXS with future extensions.

do not deliver them at last, but which work over elements, the implementation of their *next* procedure may lead up to four different variants:

(1) *Non-nested*: if none of the elements recovered from each side may contain occurrences of the same element.
(2) *Full-nested*: if elements from both sides are self-nested.
(3) *Left-nested*: if just the left side delivers elements that are self-nested.
(4) *Right-nested*: if only elements delivered by the right side are self-nested.

As a result, for a same internal node, we may have several implementations of the *next* algorithm. All of them have been designed and implemented by considering the subset of XPath addressed in this work. A detailed analysis of each version for all the operators is described in Cerdeira-Pena [2013].

6. EXPERIMENTAL EVALUATION

This section evaluates the experimental performance of XXS. We analyze both its compression properties (Section 6.1) and its query performance (Section 6.2). An isolated Intel®Pentium®Core `i5` 2.67GHz system, with 16GB dual-channel DDR-1200Mhz RAM was used in our tests. It ran Ubuntu 11.04 GNU/Linux (kernel version 2.6.38). The compiler used was g++ version 4.5.2 and `-O9` compiler optimizations were set.

6.1. Compression Properties

As previously mentioned in the Introduction, very few of the queriable compression tools existing in the literature have currently available source codes. To the best of our knowledge, only XGrind [Tolani and Haritsa 2002], XBzipIndex [Ferragina et al. 2006; 2009], SXSI [Arroyuelo et al. 2010] and TinyT [Maneth and Sebastian 2010] tools are accessible. From these, XGrind could not be run under the Linux system of our test machine.

Therefore, we have also validated XXS against some general text compression methods and XML conscious non-queriable compressors. The result of such a comparison is not completely fair, since none of these tools provides query support. Still, these compressors serve as a reference to evaluate the compression performance of XXS.

Besides compressors, we have also benchmarked some of the best state-of-the-art solutions supporting XPath, whose query performance will be analyzed in Section 6.2. In particular, we have considered the space usage of MonetDB/XQuery and Qizx/DB.

We have divided the overall set of solutions tested into three main groups, to provide a comprehensive but clear discussion:

— *General text compressors*: we have included into this category the `SCDC` compressor, as it constitutes the back-end compression method used by the `XWT` representation, and also another word-based byte-oriented semistatic statistical compressor, `Plain Huffman` [de Moura et al. 2000], based on Huffman codes. In addition, we have considered some well-known Ziv-Lempel based compressors, namely `gzip` (`http://www.gzip.org`) and `p7zip` (`http://www.7-zip.org`); a representative method of the PPM family, the `PPMdi` compressor; and finally, a compressor based on the the Burrows-Wheeler Transform [Burrows and Wheeler 1994], `bzip2` (`http://www.bzip.org`).
— *XML conscious non-queriable compressors*: this category also suffers from the lack of source code/binaries. Only those available could be compared. One is `XMill` [Liefke and Suciu 2000], which can be combined with the general back-end compressors `gzip`, `bzip2`, and `PPM`, leading to variants `XMillGzip`, `XMillBzip2`, and `XMillPPM`. We also compare `XMLPPM` [Cheney 2001] and `SCMPPM` [Adiego et al. 2007b] compressors, as well as the two variants of `XWRT` [Skibinski et al. 2008], which use

zlib (http://www.zlib.net) and lpaq (http://mattmahoney.net/dc), respectively, as back-end techniques. Finally, although XBzipIndex is generally classified as a queriable XML conscious compressor, it provides a very limited query support in comparison to the rest of the queriable solutions. Therefore, we have decided to include it into this category.

— *Queriable solutions*: this group covers SXSI [Arroyuelo et al. 2010], TinyT [Maneth and Sebastian 2010], MonetDB/XQuery[22] [Boncz et al. 2006] and Qizx/DB[23] [XML Mind products 2008].

For any of the tested compressors, we have used the maximum and minimum compression options whenever they exist. We also remark that, in the case of pure compression methods, the analysis of their compression properties includes the compression ratio and the compression and decompression times. In turn, for the queriable approaches, we have measured the global size of the representation created to allow query evaluation[24], as well as the construction times.

6.1.1. Document Corpus. We have collected a corpus of 33 documents selected from multiple data sources. Table III summarizes their main properties: name, size in MBytes (Size), maximum structure depth (MaxDepth), and both number of different words of each vocabulary (VTags, for start-tags and end-tags; VAttributes, for attribute names; VContent, for text content; and VNSearch, for comments and processing instructions) and total number of words of the document that fall into each of them (see #Tags, #Attributes, #Content, and #NSearch).

6.1.2. Results.

Compression ratios. Figure 10 shows the compression ratios[25] (in % with respect to the original document size) achieved by each of the compared solutions[26]. We have used different color ranges to make clear the distinction among the three main groups in which tools have been categorized. Regarding our proposal, we have distinguished two different compression ratios, marked as 'XWT' and 'XXS'. Recall that the XXS compression format builds on the XWT data structure. Therefore, we denote as 'XWT' the space needed just to represent the XML document using the XWT representation. In turn, 'XXS' stands for the XWT plus the waste of extra space needed to perform an efficient query evaluation, including that used for the structure of partial counters to speed up *rank* and *select* operations over the XWT byte sequence, and also that needed for the succinct tree representation of the balanced parentheses data structure. We have also considered the space used to maintain the vocabularies of words into hash tables. In this way, we will use 'XWT' values for comparisons with general compression methods, and XML conscious non-queriable compressors, while 'XXS' values will be compared against queriable solutions.

As it can be observed, XWT represents a document within 30%-40% of its original size, while XXS just amounts (in general) to an additional 4%-8% of extra space over

[22]We used version Oct2010-SP1 of MonetDB, that includes version 4.40.3 of MonetDB4 server and version 0.40.3 of the XQuery module.

[23]We used Qizx/DB free edition, version 4.2.

[24]Regarding space properties, TinyT deserves a special mention. This tool was initially devised as a structural index for XML (thus just considering XML documents structure) aimed to allow fast evaluation of specific structural XPath *count* queries. For such operations, the corresponding indexes are minuscule (tipically, less than 1% of the original XML documents size). However, to support *serialization*, additional structures must be added, to store attribute and text values. In Figure 10 we show the total size of the representation that allows TinyT both to *count* and to *serialize* query results.

[25]Missing values indicate that a tool failed to compress/decompress the document.

[26]We use -f and -b to represent the *fast* and *best* variants of a compressor, respectively.

Table III. Document properties

	Size (MB)	MaxDepth	VTags	VAttributes	VContent	VNSearch	#Tags	#Attributes	#Content	#NSearch
XMark1	55.32	12	148	9	85,441	12	1,665,820	191,160	9,276,986	13
XMark2	115.76	12	148	9	132,359	12	3,470,166	397,928	19,384,255	13
XMark3	513.96	12	148	9	417,309	12	15,381,746	1,762,307	85,916,582	13
XMark4	1,029.18	12	148	9	757,852	12	30,749,422	3,525,025	171,832,697	13
Dblp2008	282.42	6	70	6	1,750,576	14	13,856,520	1,426,867	60,222,798	17
Dblp2012	961.75	6	70	9	4,525,940	14	47,888,064	6,082,270	214,012,325	17
Psd	683.64	7	128	7	3,142,459	9	42,611,636	1,052,770	105,568,992	9
Medline1	121.02	7	156	5	266,168	0	5,732,160	138,315	16,490,261	0
Medline2	593.14	7	164	15	894,702	14	28,478,436	4,436,417	87,413,949	15
Medline3	877.32	7	166	16	1360745	14	40,199,504	6,468,566	131,882,636	15
Alfred	74.16	5	120	0	75,630	14	4,089,784	0	8,105,935	17
Baseball	0.64	6	92	0	3149	0	56,612	0	60,897	0
Lineitem	30.80	3	36	1	39,593	0	2,045,952	1	3,411,432	0
Mondial	1.78	5	46	32	19,086	30	44,846	47,423	321,201	33
Nasa	23.89	8	122	9	77,687	0	953,292	56,317	4,180,538	0
Shakespeare	7.53	7	44	0	28,346	9	359,380	0	1,505,075	9
Swissprot	112.76	5	170	14	500,909	0	5,954,062	2,189,859	23,166,916	0
Treebank	85.42	36	500	1	1,979,256	0	4,875,332	1	10,439,446	0
USHouse	0.51	16	86	21	5,179	14	13,424	2,732	82,414	15
TCSD-normal	107.18	8	48	1	613,408	33	5,499,502	7,333	22,129,473	37
DCSD-normal	105.37	8	100	3	663,514	33	4,485,398	150,000	14,547,468	37
Uniprot1	434.99	6	144	39	1,061,320	14	17,587,730	11,364,588	89,110,893	15
Uniprot2	716.00	6	144	39	1,608,280	14	28,999,340	18,671,115	146,563,011	15
EXI-Array	22.06	10	94	17	94,951	27	453,046	226,550	3,600,182	33
EXI-Factbook	4.04	5	398	0	28,013	39	110,906	0	604,601	54
EXI-Invoice	0.93	7	104	7	16,748	9	30,150	14,060	109,538	9
EXI-Weblog	2.53	3	24	0	1,260	0	186,870	0	435,894	0
EnwikiNews	69.42	5	40	7	311,877	0	809,304	35,000	15,416,589	0
EnwikiQuote	124.27	5	40	7	412,082	0	525,910	23,837	29,155,406	0
EnwikiTionary	556.61	5	40	7	3,479,730	0	16,770,268	726,129	104,853,291	0
EnwikiVersity	81.40	5	40	7	300,349	0	991,678	43,621	18,830,566	0
EnwikiAbstract1	660.56	5	18	1	540,589	0	28,327,694	3,811,222	140,817,649	0
EnwikiAbstract2	327.96	5	18	1	420,168	0	13,692,938	1,714,361	70,280,032	0

the XWT basic representation. More precisely, we use about 3%-4% of extra space in the *rank/select* structures and the balanced parentheses succinct representation. The other 1%-4% is used to maintain the vocabularies in hash tables.

Regarding general text compressors (see the values marked in black in Figure 10), if we compare XWT compression ratios with SCDC, which constitutes the base of the XWT compression scheme, we note that XWT needs, on average, about 3%-4% more space. This is the price of reserving special first bytes for the codewords of separate alphabets. The same small difference is kept with respect to Plain Huffman (PH). In comparison with the rest of the general text compressors, and also the XML conscious non-queriable solutions (see the pink marks in Figure 10), differences may vary for each technique, yet almost all of them achieve better compression ratios than XWT. We remind, however, that these compression formats do not support queries on the compressed documents, and thus they just optimize for space.

A fairer analysis arises from the comparison of XXS with other queriable solutions (see the values depicted in green in Figure 10). Recall that, in this scenario, we must consider the values corresponding to the 'XXS' label, which include the overall space usage of our proposal. As shown in Figure 10, our tool is by far the system with the best compression ratios. In particular, XXS needs between 2 and 5 time less space than any other queriable solution.

Therefore, it is clear that XXS achieves compression ratios much closer to those obtained by non-queriable solutions, than to queriable tools. The most striking feature is that, even using such a little amount of space, our tool provides query evaluation capabilities like those of the significantly bulkier queriable ones.

Compression and decompression times. With regard to time measures[27], if we focus on the performance of general text compressors (see the plots at the top of Figure 11) we notice that XWT needs, in general, more time to compress the input data than both SCDC and PH codes, mainly due to the more complex parsing we perform to handle the XML features. On the other hand, decompression times are not affected, and even improve in many cases. From the behavior of the rest of the general text compressors, we can infer that XWT outperforms both compression and decompression times of virtually all of them.

With respect to the XML conscious non-queriable compressors (see bottom left of Figure 11), we can see that all these techniques require significantly more time to compress than XWT[28]. They are also much slower than their general-purpose counterparts. XWT is also unbeaten in decompression time (see bottom right of Figure 11).

Finally, we compare in Figure 12 the construction times of the queriable solutions. The construction times of XXS match the time to build the XWT representation and to store it on disk, since the additional structures used for efficient searching[29] are created on-the-fly when the data structures are loaded from disk. The results show that XXS and MonetDB/XQuery are the queriable alternatives fastest to build, achieving a speed over 10 MB/sec, whereas Qizx/DB, SXSI and TinyT are usually slower. In particular, the construction speed of the last two is usually below 1 MB/sec.

6.2. Query Evaluation Performance

To illustrate the behavior of XXS in query evaluation, we have compared its performance with a group of well-established queriable solutions, namely MonetDB/XQuery, Qizx/DB, and SXSI (see Section 6.2.2). Although both XBzipIndex and TinyT may be also categorized as queriable tools, they are intended for very specific XPath queries. Hence, we have devoted a separate section (Section 6.2.3) to analyze them.

Additionally, we have decided not to include in this study comparisons with streaming XPath engines (e.g., GCX and SPEX) or in-memory processors (e.g., Galax and Saxon). Such a comparison is hardly fair since, in the first scenario, streaming processors need to parse the whole XML document at each run. For instance, the SPEX streaming processor runs about 475 times slower than XXS. In turn, the limitation of in-memory processors arises from the high times required to build the in-memory representation, prior to evaluating each query. For example, this makes Saxon run about 125 times slower than XXS. Moreover, this kind of tools usually represents XML data using machine pointers, which blow up memory consumption. For example, Saxon needs 4-5 times the size of the original XML documents used in our experiments.

It is interesting to mention how the chosen tools perform on the tests designed by Gottlob et al. [2005] to detect algorithms that are exponential on the query size. While the times of XXS and MonetDB increase only linearly with the query size, those of SXSI and Qizx/DB increase exponentially.

[27]Figures 11 and 12 use the same legends and colors as Figure 10.
[28]Except XMillGzip compressor with the minimum compression options, which gets similar compression times to those of our tool.
[29]The *rank/select* structures and the balanced parentheses representation.

6.2.1. Query Test Bed. The experimental framework for query evaluation has been designed to be tested over any of the `XMark` documents presented in Section 6.1.1. These are files generated with *xmlgen*, an XML data generator modeling an auction website that has been developed inside the *XMark Project* (`http://monetdb.cwi.nl/xml`)[30]. We have developed a comprehensive query test bed that evaluates the whole practical subset of XPath discussed in Section 4.1, and that aims to test the efficiency, scalability and stability of the analyzed systems. Queries have been divided into four main groups, described next (see Figures 13 and 14):

— `Structural (Q01-Q21)`: these queries are taken from the *XPathMark* benchmark[31], which simulates realistic query needs of a potential user of an auction site. We have taken the queries related to the practical subset of XPath addressed in this work, that is, all the queries covering the *forward* and *reverse* XPath axes, using as node tests either a tag/attribute name or the wildcard '*', and that admit the use of predicates, in combination with conjunctive and disjunctive boolean operators. We have also included some additional queries, created ad-hoc, exhibiting the same properties.
— `Wildcards (Q22-Q42)`: one of the most challenging scenarios for query evaluation is that posed by queries involving a sequence of steps over the wildcard '*', due to the potentially high number of intermediate results that may be generated (e.g., `/book/*/*//*/image`). This group of queries aims to validate the performance of the systems in these situations.
— `Names (Q43-Q58)`: these queries aim to seek the occurrences of specific elements and attributes chosen at random. We also regard the special queries that search for any element (Q43) or attribute appearance (Q54).
— `Text (Q59-Q73)`: previous groups of queries are composed by purely structural constraints. This group is designed to cover examples of typical queries that a user could formulate by using the `contains` and `equal` functions, applied over either an element content or an attribute value. They include both word and phrase patterns.

6.2.2. Comparison with Full-fledged Solutions. We have run the set of queries described on the documents `XMark2` and `XMark4` of our collection (see Table III). For each query of the test bed we have measured the running times (in milliseconds) of the main search operations, namely *count*[32], *materialize* (locate) and *materialize+serialize*[33] (display) the results. We have used the systems timing reports, and kept the best of five runs. For MonetDB/XQuery, times are given by option `-t` of the client program, `mclient`. The server is properly exited and restarted before each group of five runs. For Qizx/DB, we used option `'-r 2'` of the command line interface to run twice each individual run (the second one being always faster). We ignore the time to load the index into main memory, in any system.

We must also remark that in case of Qizx/DB it is not possible to isolate materialization times, so it was only compared in the other two scenarios. On SXSI, some of the queries could not be run, as it does not support `following`, `attribute` or reverse axes.

The running times for the complete set of queries presented in Section 6.2.1 is available in Cerdeira-Pena [2013][34]. For conciseness, here we will provide a general

[30]We have focused on these documents of the data set, as the *XMark Project* has been acknowledged as a reference for benchmarking XML data.
[31]`http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark`
[32]In this case, queries are run by adding the XPath `count` function to each one. For instance, a query such as `//open_auction //price` will result into `count(//open_auction//price)`.
[33]Results are serialized to the `/dev/null` device in order to discard the output.
[34]`http://lbd.udc.es/Repository/Thesis/1366361229174_PhD_acerdeira.rar`.

overview, discussing the most important facts about the performance of the systems. Figures 15 to 26 illustrate, for each group of queries, the percentage of queries (computed over the total number of queries of the group) for which each system obtained the best running times. They also include more detailed reviews of the results for a selection of queries most representative of the overall results. These graphs should be read as follows. For each query pointed at the bottom of the graph, vertical bars represent the relative running times of the systems with respect to the tool that reached the best time (whose score is always 100%). In addition, we also provide at the top of each bar the actual running time of the query in milliseconds (or seconds, if it is suffixed with an "s"). Missing values mean either that a query is not supported by a system or that the query did not complete within reasonable time.

Group 'Structural'. Figures 15 to 17 depict the performance of the systems for the group of queries `Structural` over the documents `XMark2` and `XMark4`. For counting and materializing (see Figures 15 and 16), the results show that XXS performs on par and even better than the other solutions, achieving the best running times in most queries. We also note that, in those cases, both XXS and SXSI scale well, whereas MonetDB/XQuery does not: It performs better over `XMark2`, but its performance degrades over the larger `XMark4`. The opposite happens to Qizx/DB, although this system performs poorly in general.

With respect to materializing plus serializing times, Figure 17 shows that the best results are usually obtained by MonetDB/XQuery and SXSI when dealing with the small document instance, `XMark2`. Yet, MonetDB/XQuery does not perform so well for the larger document, `XMark4`, whereas XXS and SXSI scale well. Again, Qizx/DB does not obtain any outstanding result. The reason why XXS does not compete as well as for counting or materializing is intrinsic to its goals: it maintains the data in compressed form, and thus there is a time penalty for decompressing it.[35] The other systems, instead, can afford to maintain a copy of the original text and thus can output any portion of the data with much less effort. Even SXSI, which uses compressed representations, maintains a plain copy of the text to enable fast serialization of results. Notwithstanding, we recall that another relevant feature of XXS is its ability to obtain the results upon user demand, as in most text search engines. Cerdeira-Pena [2013] analyzes the performance of XXS in a scenario where the results are consumed gradually, measuring the times to deliver a first batch of 50 results per query. In most cases, those results are reported in less than a millisecond.

Group 'Wildcards'. These queries aim to evaluate the robustness of the systems on queries involving several steps over the '*' wildcard, and in particular, the benefits of the wildcard optimizations we designed for XXS during the construction of the query execution plan (see Section 4.3). In this case, results are shown in Figures 18 to 20 just for the larger document, as all the systems behave similarly on the smallest one[36]. As it can be observed, XXS clearly overcomes the rest of the systems for counting and materializing. This is not always the case when results are to be displayed, as before.

Group 'Names'. This is composed of basic queries that count, materialize and serialize the occurrences of a given element or attribute. In the general XXS evaluation scheme, to obtain the number of results of a given query, the query must first be materialized (that is, its results must be located). However, for this group of queries, the

[35] Indeed, the time taken by XXS to serialize the results shadows the query processing time itself.
[36] With the only exception of the serialization scenario, where MonetDB/XQuery gets the best results on half of the queries, mainly in detriment of SXSI and Qizx/DB.

count operation is performed more efficiently by using a simple *rank* operation[37]. As shown in Figure 21[38], XXS is the fastest by far in this scenario (just requiring some microseconds). Yet, for materializing and serializing (see Figures 22 and 23), we cannot take advantage of that procedure. In any case, we notice that these kind of queries are also subject to optimization in the other systems. Observe that, for instance, MonetDB/XQuery, which usually worsens on the largest document, obtains the best materialization times over `XMark4` for an important set of queries (see queries Q47 to Q53 in Figure 22).

Group 'Text'. To evaluate the performance of the systems over queries involving a text function, we used the *Full text* extension of XQuery [W3C 2011] available in the tested version of Qizx/DB, and rewrote some of the queries of this group to make them as efficient as possible, while preserving the semantics of the original ones. In particular, we used the `ftcontains` text function instead of the standard `contains`, as it is more efficient. For MonetDB/XQuery, the included PF/Tijah text index [List et al. 2005] also supports some full-text capabilities. However it does not provide an optimized version of the `contains` operator, hence we used the standard one, that relies on string conversions. Finally, we note that the `contains` and `equal` implementations of SXSI do not support text searches over phrases spanning more than one text node.

Figures 24 to 26 present the results obtained for the group of text oriented queries. As it can be seen, XXS performs on par with SXSI, and with MonetDB/XQuery for tests over `XMark2` (as none of them actually stands out from the other), all of them outperforming Qizx/DB[39]. However, in case of `XMark4`, MonetDB/XQuery becomes much slower, while both XXS and SXSI scale well. As before, Qizx/DB performs better on the larger document, and in particular it stands out for counting on `XMark4`.

An important fact is that text oriented queries turn out to be much more selective than the groups of structural based queries, in terms of number of results produced. Hence, XXS materialization plus serialization times are not as affected by the times required to decompress the words before outputting them, as happened before.

To summarize, according to the experimental evaluation performed, we can highlight the three following features as the base properties that define the global behavior of our system, leaving it in a cutting edge position compared with some of the best-known state-of-the-art solutions supporting XPath:

(1) XXS uses between 2 and 5 times less space than any of the compared solutions.
(2) XXS is, in general, the fastest alternative for *counting* and *materializing* queries.
(3) XXS is not the fastest one at *displaying*. Yet, it is not far from the other alternatives, and moreover its underlying lazy evaluation scheme allows serializing the results immediately, delivering them upon user needs.

6.2.3. Comparison with Indices Offering Limited Support. The tools analyzed in this section provide a limited XPath query support. For instance, in case of XBzipIndex, solely full-specified paths of the form $//x_1/\ldots/x_n$ and $//x_1/\ldots/x_n[contains(., \gamma)]$, where x_1 and x_n denote tag/attribute names[40], and γ is an arbitrary string, are supported. For TinyT, it is not necessary to set the complete path, as this tool allows one to use the '*' wildcard[41]. Nonetheless, TinyT only supports `child` and `descendant` axes, while filters

[37] Recall that, to count number of occurrences of a given word, it is just necessary to compute how many times the last byte of its codeword appears in the corresponding node of the XWT.
[38] Again, we only show the results for `XMark4`; the same conclusions are obtained for `XMark2`.
[39] With the exception of MonetDB/XQuery for some queries.
[40] The available binaries of this tool do not admit the use of attributes.
[41] Similarly to XBzipIndex, TinyT binaries do not handle attributes.

are not allowed (nor data value comparisons). Therefore, just a small group of queries from the complete query test bed presented in Section 6.2.1 could be run over each of these two tools[42].

Regarding XBzipIndex, results show that this tool performs much slower than any other solution, for any query. TinyT deserves a more detailed discussion[43]. Recall that this tool was initially conceived to speed up count operations over structural XPath queries. In fact, for the *counting* scenario, the set of queries analyzed from groups `Structural` and `Wildcards` show that TinyT obtains better results than XXS (which turned out to be the best system from the comparisons of Section 6.2.2) over virtually all the tested queries, yet the time differences are in the same order of magnitude or just one order higher. The comparison changes when considering the queries of the group `Names` (for which the *count* operation is optimized in XXS). In this group, XXS has no competitors.

Before reviewing *materialization* and *materialization+serialization* operations, we must remark that TinyT does not allow materializing the results, and that serialization[44] also avoids materialization. Hence, just the second scenario could be analyzed over TinyT. Furthermore, the results obtained for such a situation can not be compared straightforwardly with the rest of the tools, since they do not include materialization times. In any case, for the subset of queries from groups `Structural` and `Wildcards`, TinyT improves the best results, but within the same order of magnitude. This is not so homogeneous on the queries of group `Names`, as SXSI still exhibits the best running times for several queries.

As shown, even in comparison with more restricted, specialized indices, the performance of XXS remains outstanding.

7. CONCLUSIONS AND FUTURE WORK

As the adoption of the XML standard spreads over more and more areas related to information retrieval, data manipulation and knowledge representation, the challenges of efficiently operating it become more crucial. Two of the most striking problems are (1) the complexity of its standard query languages, XPath and XQuery, and the difficulty of supporting them efficiently, and (2) the amount of space required by the representation of the XML data and its indexes, which also impacts the time performance. Despite much recent research, one can safely say that there are no available, practical, and scalable solutions properly addressing these two challenges simultaneously.

In this paper we have presented XXS, a tool that tackles both issues through the use of a compressed self-indexed representation of the XML data. This representation encodes the XML data in a form that reduces space and at the same time enables powerful queries on it. XXS is aimed at semistructured natural language text collections, and to be operated in main memory. In our experiments, it reduces the XML data to 35%-50% of its original size, and within this compressed size it efficiently supports a large subset of the XPath query language. While bare compressors can achieve better compression ratios, XXS uses 2-5 times less space than any other tool we are aware of that can support a reasonable subset of XPath. It also requires less time to build the representation. This makes XXS an attractive alternative to manipulate larger XML collections in main memory.

[42]In particular, queries Q01, Q44-Q53, and Q61, for XBzipIndex; and queries Q01-Q03, Q22-Q27, Q38-Q42, and Q43-Q53, for TinyT. The specific running times for those experiments are available at `http://vios.dc.fi.udc.es/acerdeira`.

[43]The same conclusions apply for both `XMark2` and `XMark4` documents.

[44]Like SXSI, TinyT also maintains a copy of the text for fast data extraction.

The query evaluation engine of XXS builds on the following principles: (i) efficient implementation of some core operations using the self-index data structure, (ii) query optimization adapted to the cost model of the self-index, (iii) translation of XPath operations to restrictions on segments covered by the structures, (iv) lazy evaluation with results flowing bottom up in the query parse tree, strengthened with (v) skipping restrictions that flow top-down. Our comprehensive experimental results highlight the good performance of XXS. Most of the times, it performs better than the best current systems supporting XPath queries, both for counting and for materializing queries. Only when serialization is involved, the performance of XXS is degraded due to the need to decompress the data (whereas other systems can maintain the text in plain form). Yet, the results are still competitive. Moreover, the lazy evaluation capability of XXS allows it to obtain the results upon user demand, which is very valuable when the results are directly consumed by persons. In this scenario, XXS can report, within one millisecond in most queries, a first batch of, say, 50 query results, and continue producing the rest while the others are being analyzed by the user.

As a general conclusion, we can say that our proposal requires little space, provides efficient XPath querying capabilities, and displays a robust and scalable behavior. These features leave XXS without competitors with comparable query evaluation performance while using similar space. The usefulness of XXS in real-life scenarios is also being demonstrated in a current project for the integration of XXS within the *Miguel de Cervantes Digital Library* (`http://www.cervantesvirtual.com`), the largest repository of digitalized texts from the Spanish literature.

We plan to extend the subset of XPath targeted in this work, in order to include XPath extensions such as inequalities and positional predicates, and eventually aim at supporting XQuery. As XPath constitutes the core of the XQuery language, we intend to exploit the efficient querying capabilities of XXS to solve FLWOR expressions.

Another quite interesting future line of research is to introduce ranking of results, which is essential in an Information Retrieval scenario. This requires the adoption of a relevance measure that is compatible with a hierarchical text model, which is a research topic by itself [Lalmas 2009]. The suitability of the XWT structure for ranked document retrieval (a simplified case where the structure consists of plain text documents) with simple conjunctive and disjunctive text queries has already been demonstrated [Brisaboa et al. 2012]. This suggests that the XXS data organization may be suitable for the more complex task of ranked retrieval on structured text.

APPENDIX

We include a glossary of the most frequent acronyms used along the paper.

Streaming XPath engines	
GCX	Stream processor that uses a buffer management scheme in combination with *document projections*.
SPEX	XPath query evaluator over XML data streams based on pushdown transducers.

Indexed XPath solutions	
In-memory engines	
Galax	Query processor over in-memory XML data model instances built at runtime.
Saxon	Main-memory processor based on in-memory DOM/DTM XML representations.
Database systems	
MonetDB/XQuery	RDBMS providing full support of XQuery.
Qizx/DB	Native XML database system fully supporting XQuery, and its full-text extension.
General text compressors	
Bzip2	Compressor based on the Burrows Wheeler Transform (BWT).
ETDC	*End-Tagged Dense Code*: the simplest word-based bytewise encoder from the Dense Code family.
Gzip	Ziv-Lempel compressor based on LZ77 technique.
p7zip	Ziv-Lempel compressor based on LZMA algorithm.
PH	*Plain Huffman*: a word-based byte-oriented semistatic statistical compressor, based on Huffman codes.
PPMdi	Statistical adaptive compressor from the PPM (*Prediction by Partial Matching*) family.
(s, c)-DC / SCDC	(s, c)-Dense Code: dense code generalization of ETDC.
Non-queriable XML compressors	
SCMPPM	*Structure Context Modeling* (SCM) variant based on PPM compression techniques.
XMill	First approach to XML conscious compression. Structure and data containers are separately compressed.
XMLPPM	Streaming XML compressor based on the *Multiplexed Hierarchical Modeling* (MHM) technique that combines SAX encoding and PPM compression scheme.
XWRT	Dictionary-based compression technique that applies similar ideas to XMill.
Queriable XML compressors	
SXSI	Up-to-date proposal for compressed indexing of XML documents.
TinyT	Structural self-index for XML based on grammar-based tree compression.
XBzipIndex	Compressed and searchable implementation of the XML Burrows Wheeler Transform (XBWT).
XGrind	First XML conscious queriable compressor able to support queries over the compressed form. It does not separate structure from data content.
XXS	Our proposal: a self-index for efficient XPath evaluation within the space of the compressed text.
Other structures	
BP	*Balanced parentheses* succinct tree representation.
WTBC	Codeword bytes rearrangement of a natural language text compressed with any word-based byte-oriented semistatic statistical encoding scheme.
XWT	Compressed self-indexed XML representation (core part of XXS).

ACKNOWLEDGMENTS

Funded in part by MICINN grants (PGE and FEDER) TIN2009-14560-C03-02 and TIN2010-21246-C02-01, Xunta de Galicia grants (co-funded with FEDER) GRC2013/053 and CN 2012/211, and MINECO grants (co-funded with CDTI and GAIN) CDTI EXP 00064563 and ITC-20133062 (for the Spanish group); and by Fondecyt grants 1-080019 and 1-110066, Chile (G.N.). We thank Felipe Sologuren and Kim Nguyễn for useful discussions.

REFERENCES

ADIEGO, J., NAVARRO, G., AND DE LA FUENTE, P. 2007a. Lempel-Ziv compression of highly structured documents. *Journal of the American Society for Information Science and Technology 58,* 4, 461–478.

ADIEGO, J., NAVARRO, G., AND DE LA FUENTE, P. 2007b. Using structural contexts to compress semistructured text collections. *Information Processing and Management 43,* 3, 769–790.

ARION, A., BONIFATI, A., MANOLESCU, I., AND PUGLIESE, A. 2007. XQueC: A query-conscious compressed XML database. *ACM Transactions on Internet Technology 7,* 2.

ARROYUELO, D., CLAUDE, F., MANETH, S., MÄKINEN, V., NAVARRO, G., NGUYEN, K., SIRÉN, J., AND VÄLIMÄKI, N. 2010. Fast in-memory XPath search using compressed indexes. In *Proc. of the 26th IEEE International Conference on Data Engineering (ICDE)*. 417–428.

BAEZA-YATES, R. A. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley Longman.

BONCZ, P. A., GRUST, T., VAN KEULEN, M., MANEGOLD, S., RITTINGER, J., AND TEUBNER, J. 2006. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 479–490.

BOURRET, R. 2009. Going native: Use cases for native XML databases. `http://www.rpbourret.com/xml/UseCases.htm`.

BRISABOA, N. R., CERDEIRA-PENA, A., AND NAVARRO, G. 2009. A compressed self-indexed representation of XML documents. In *Proc. of the 13th European Conference on Digital Libraries (ECDL)*. LNCS 5714. 273–284.

BRISABOA, N. R., CERDEIRA-PENA, A., NAVARRO, G., AND PEDREIRA, O. 2012. Ranked document retrieval in (almost) no space. In *Proc. of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*. 155–160.

BRISABOA, N. R., FARIÑA, A., LADRA, S., AND NAVARRO, G. 2012. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval 15,* 6, 527–557.

BRISABOA, N. R., FARIÑA, A., NAVARRO, G., AND PARAMÁ, J. R. 2007. Lightweight natural language text compression. *Information Retrieval 10,* 1, 1–33.

BURROWS, M. AND WHEELER, D. 1994. A block-sorting lossless data compression algorithm. In *Tech. Rep 124, Digital Equipment Corporation*.

CERDEIRA-PENA, A. 2013. Compressed self-indexed XML representation with efficient XPath evaluation. Ph.D. thesis, Department of Computer Science, University of A Coruña, Spain.

CHENEY, J. 2001. Compressing XML with multiplexed hierarchical PPM models. In *Proc. of the 11th Data Compression Conference (DCC)*. 163–172.

CHENG, J. AND NG, W. 2004. XQzip: Querying compressed XML using structural indexing. In *Proc. of the 9th International Conference on Extending Database Technology (EDBT)*. 219–236.

CLEARY, J. AND WITTEN, I. H. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communication 32,* 4, 396–402.

CULPEPPER, J. S. AND MOFFAT, A. 2005. Enhanced byte codes with restricted prefix properties. In *SPIRE*. 1–12.

DE MOURA, E. S., NAVARRO, G., ZIVIANI, N., AND BAEZA-YATES, R. A. 2000. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems 18,* 2, 113–139.

FERNÁNDEZ, M. F., SIMÉON, J., CHOI, B., MARIAN, A., AND SUR, G. 2003. Implementing XQuery 1.0: The Galax experience. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*. 1077–1080.

FERRAGINA, P., LUCCIO, F., MANZINI, G., AND MUTHUKRISHNAN, S. 2006. Compressing and searching XML data via two zips. In *Proc. of the 15th International World Wide Web Conference (WWW)*. 751–760.

FERRAGINA, P., LUCCIO, F., MANZINI, G., AND MUTHUKRISHNAN, S. 2009. Compressing and indexing labeled trees, with applications. *Journal of the ACM 57,* 1, 4:1–4:33.

GIRARDOT, M. AND SUNDARESAN, N. 2000. Millau: an encoding format for efficient representation and exchange of XML over the web. *Computer Networks 33,* 1-6, 747–765.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2005. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems 30,* 2, 444–491.

GROSSI, R., GUPTA, A., AND VITTER, J. S. 2003. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

HUFFMAN, D. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40,* 9, 1098–1101.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 549–554.

KAY, M. 2008. Ten reasons why Saxon XQuery is fast. *IEEE Data Engineering Bulletin 31,* 4, 65–74.

LALMAS, M. 2009. *XML Retrieval*. Morgan & Claypool Publishers.

LEAGUE, C. AND ENG, K. 2007. Schema-based compression of XML data with RELAX NG. *Journal of Computers 2,* 10, 9–17.

LEIGHTON, G., DIAMOND, J., AND MÜLDNER, T. 2005. AXECHOP: A grammar-based compressor for XML. In *Proc. of the 15th Data Compression Conference (DCC)*. 467.

LEIGHTON, G., MÜLDNER, T., AND DIAMOND, J. 2005. TREECHOP: A tree-based queriable compressor for XML. Technical report, Acadia University.

LEVENE, M. AND WOOD, P. 2002. XML structure compression. In *Proc. of the 2nd International Workshop on Web Dynamics*.

LI, W. 2003. XComp: an XML Compression Tool. M.S. thesis, University of Waterloo, Waterloo, Ontario, Canadá.

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

LIN, Y., ZHANG, Y., LI, Q., AND YANG, J. 2005. Supporting efficient query processing on compressed XML files. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC)*. 660–665.

LIST, J. A., MIHAJLOVIC, V., RAMÍREZ, G., DE VRIES, A. P., HIEMSTRA, D., AND BLOK, H. E. 2005. Tijah: Embracing IR methods in XML databases. *Information Retrieval 8,* 4, 547–570.

MANETH, S. AND SEBASTIAN, T. 2010. Fast and tiny structural self-indexes for XML. *CoRR abs/1012.5696*.

MEIER, W. 2002. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*. 169–183.

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

MUNRO, J. I. AND RAMAN, V. 2001. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing 31,* 3, 762–776.

NAVARRO, G. AND BAEZA-YATES, R. 1997. Proximal Nodes: a model to query document databases by content and structure. *ACM Transactions on Information Systems 15,* 4, 400–435.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1.

NG, W., LAM, W. Y., WOOD, P. T., AND LEVENE, M. 2006. XCQ: A queriable XML compression system. *Knowledge and Information Systems 10,* 4, 421–452.

OLTEANU, D. 2007. SPEX: Streamed and progressive evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering 19,* 7, 934–949.

OSZU, T. 2003. XBench - a family of benchmarks for XML DBMSs. `https://cs.uwaterloo.ca/~tozsu/ddbms/projects/xbench/Specification.html`.

PENG, F. AND CHAWATHE, S. S. 2005. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems 30,* 2, 577–623.

SADAKANE, K. AND NAVARRO, G. 2010. Fully-functional succinct trees. In *Proc. of the 21th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 134–149.

SAKR, S. 2009. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences 75,* 5, 303–322.

SCHMIDT, M., SCHERZINGER, S., AND KOCH, C. 2007. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *Proc. of the 23rd International Conference on Data Engineering (ICDE)*. 236–245.

SKIBINSKI, P., GRABOWSKI, S., AND SWACHA, J. 2008. Effective asymmetric XML compression. *Software: Practice and Experience 38,* 10, 1027–1047.

SUBRAMANIAN, H. AND SHANKAR, P. 2005. Compressing XML documents using recursive finite state automata. In *Proc. of the 10th International Conference on Implementation and Application of Automata (CIAA)*. 282–293.

TOLANI, P. M. AND HARITSA, J. R. 2002. XGrind: A query-friendly XML compressor. In *Proc. of the 18th International Conference on Data Engineering (ICDE)*. 225–234.

TOMAN, V. 2003. Compression of XML data. M.S. thesis, Charles University, Prague, Czech Republic.

W3C. 1998. Recommendation of Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml`.

W3C. 1999. Recommendation of XML Path Language (XPath) 1.0. `http://www.w3.org/TR/xpath`.

W3C. 2010a. Recommendation of XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20`.

W3C. 2010b. Recommendation of XML Query Language (XQuery) 1.0. `http://www.w3.org/TR/xquery`.

W3C. 2011. Recommendation of XQuery and XPath Full Text 1.0. `http://www.w3.org/TR/xpath-full-text-10`.

WANG, H., LI, J., LUO, J., AND HE, Z. 2004. XCPaqs: Compression of XML documents with XPath query support. In *Proc. International Conference on Information Technology: Coding and Computing (ITCC)*. 354–358.

WELCH, T. A. 1984. A technique for high-performance data compression. *IEEE Computer 17,* 6, 8–19.

WONG, R. K., LAM, F., AND SHUI, W. M. 2007. Querying and maintaining a compact XML storage. In *Proc. of the 16th International World Wide Web Conference (WWW)*. 1073–1082.

XML MIND PRODUCTS. 2008. Qizx XML database engine. `http://www.axyana.com/qizx`.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23,* 3, 337–343.

ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24,* 5, 530–536.

TEXT: “ MAKE EVERYTHING AS SIMPLE AS POSSIBLE BUT NOT SIMPLER”

SYMBOL	FREQUENCY	CODE
AS	2	b_1
POSSIBLE	1	b_3b_1
EVERYTHING	1	b_4b_1
SIMPLE	1	b_4b_2
MAKE	1	b_5b_1
NOT	1	b_5b_2
BUT	1	$b_3b_4b_2$
SIMPLER	1	$b_4b_5b_1$

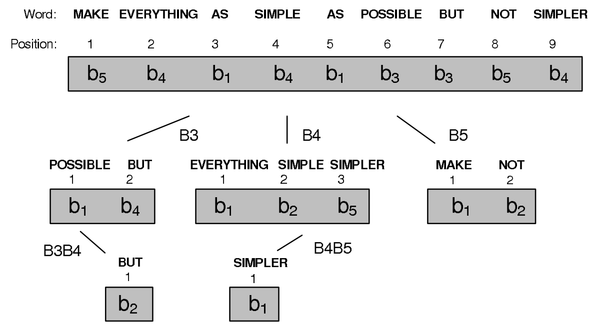


Fig. 1. Example of WTBC structure.

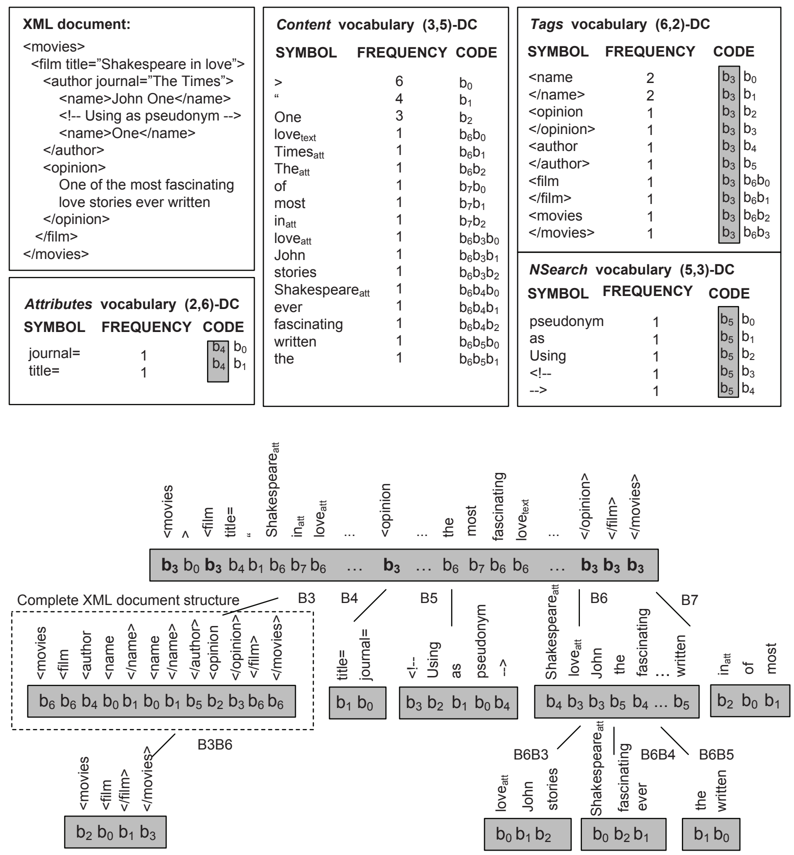


Fig. 2. Example of XWT structure built from an XML document.

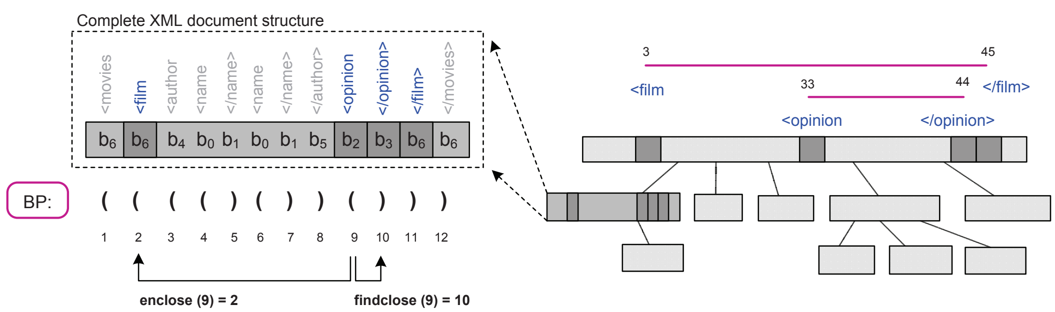


Fig. 3. Correspondence between the root of the XWT subtree storing the document structure and a balanced parentheses representation.

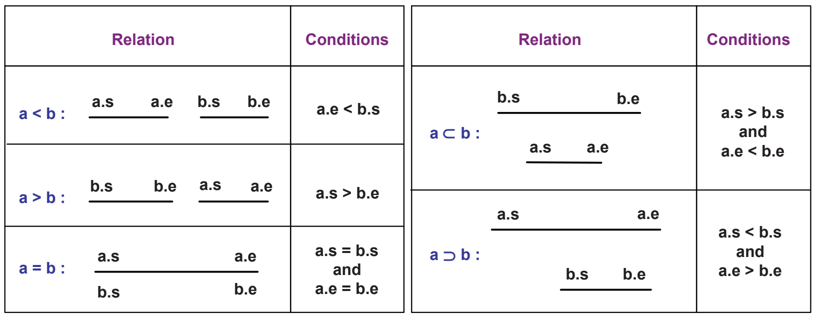


Fig. 4. Segment relationships.

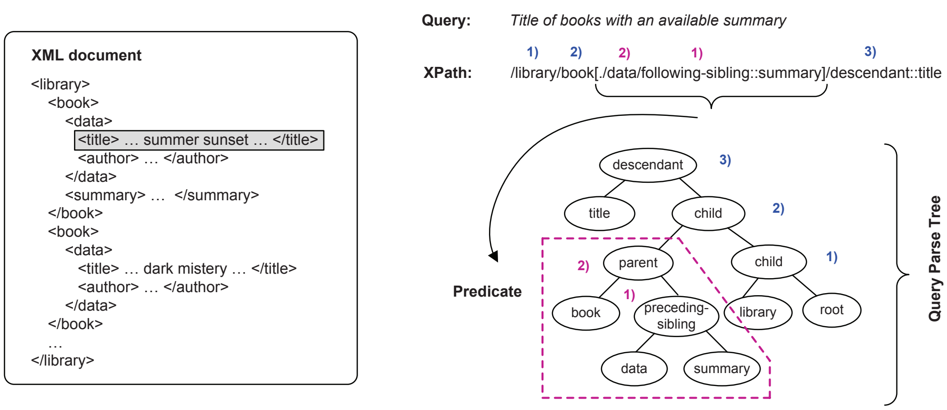


Fig. 5. Example of *query parse tree*.

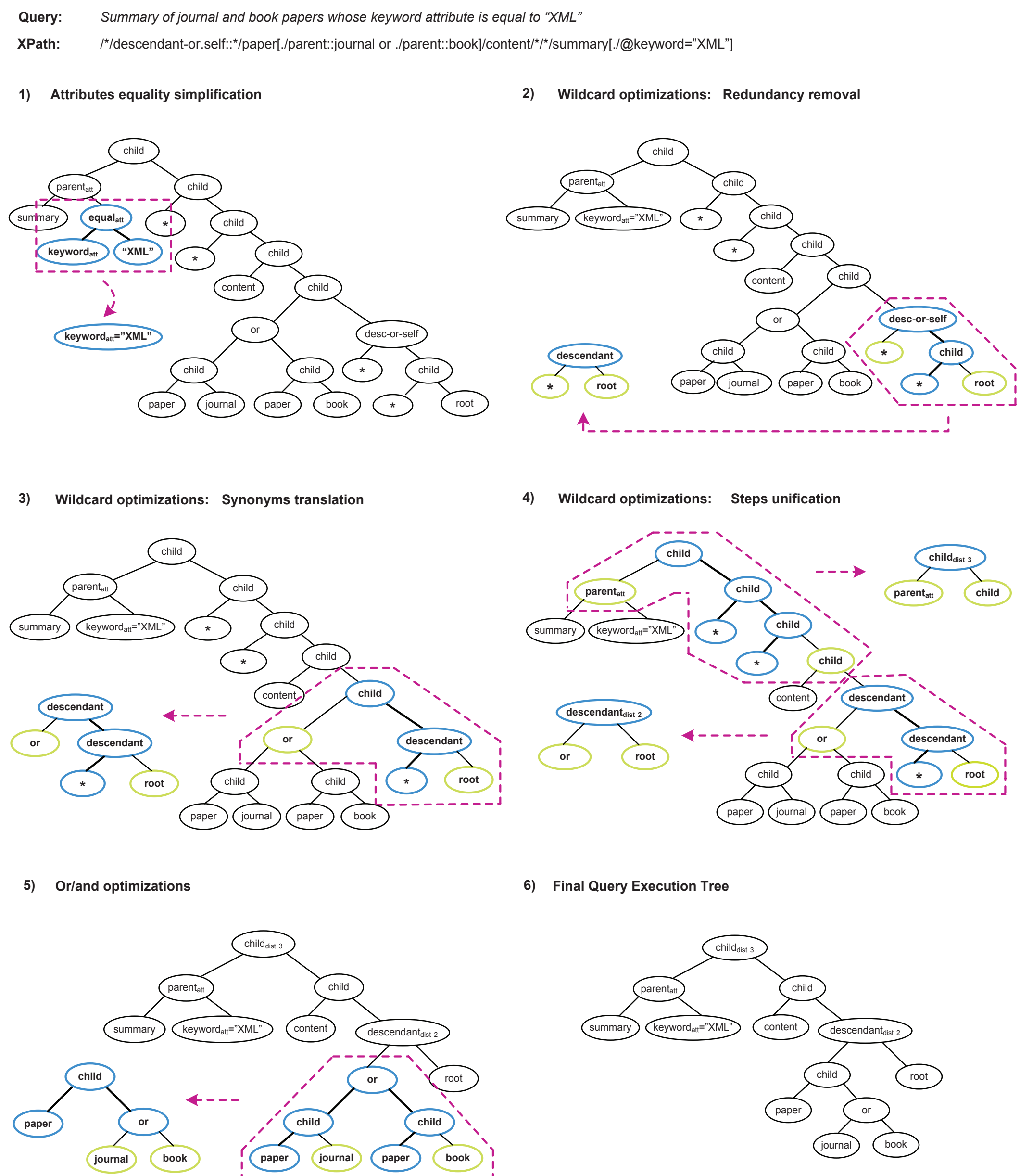
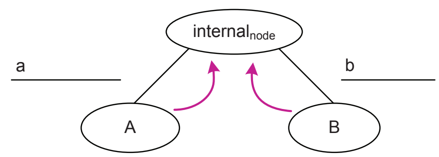


Fig. 6. Optimizations applied over a *query parse tree* until reaching the final *query execution tree*.

$\text{Internal}_{\text{node}}$	Relation
ancestor	$a \supset b$
descendant	$a \subset b$
parent	$a \supset b$ *
child	$a \subset b$ *
self	$a = b$

$\text{Internal}_{\text{node}}$	Relation
following	$a > b$
preceding	$a < b$
following-sibling	$a > b$ †
preceding-sibling	$a < b$ †

† Segments must share the same parent, in addition



* Additional validation of the segments depth
→ depth(a) = depth(b) – 1 (parent)
→ depth(a) = depth(b) + 1 (child)

Fig. 7. Relations that compared segments must hold to satisfy the semantics of different XPath axes.

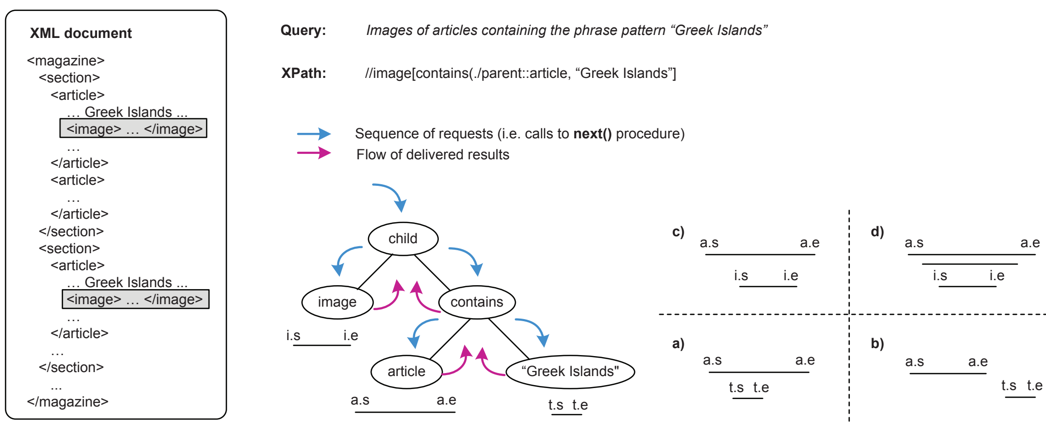


Fig. 8. General query evaluation scheme.

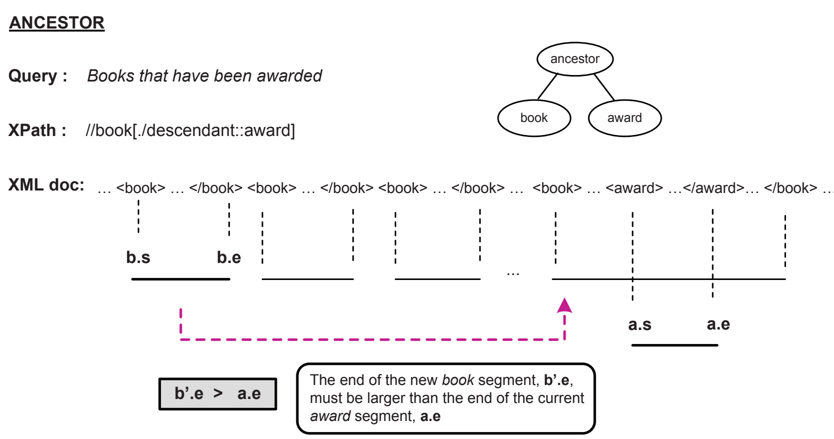


Fig. 9. Skipping of segments.

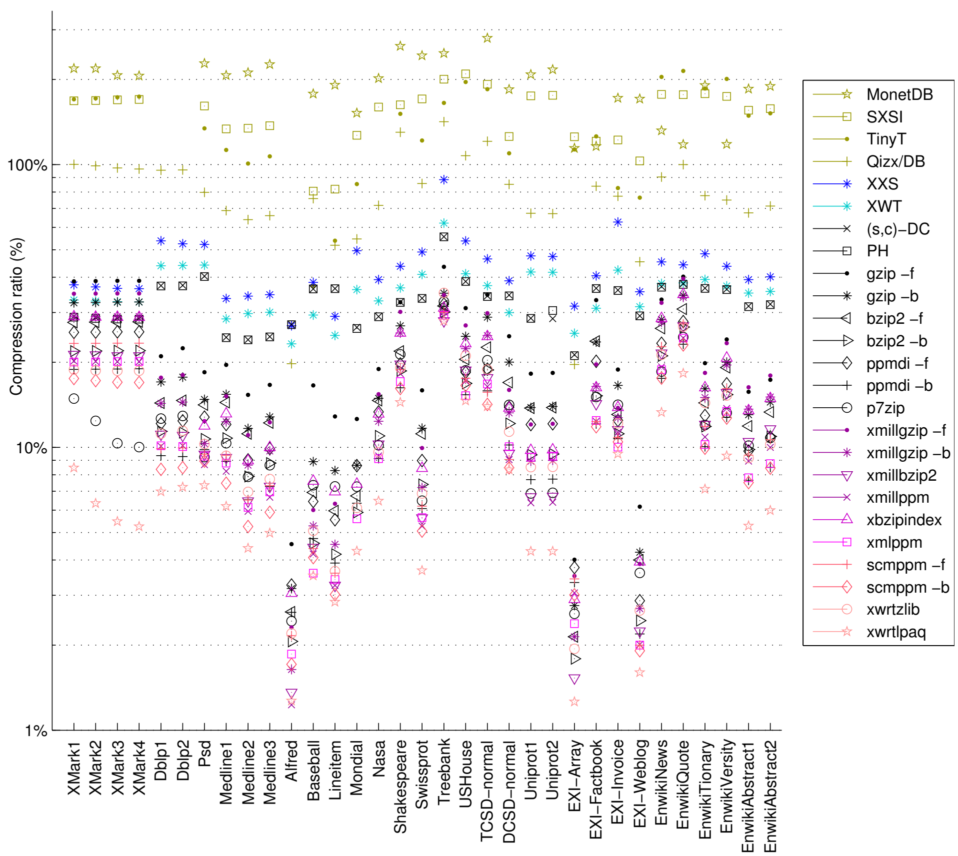


Fig. 10. Compression ratios achieved by our proposal (in blue), general text compressors (in black), XML conscious non-queriable compressors (in pink), and queriable tools (in green) over different XML documents.

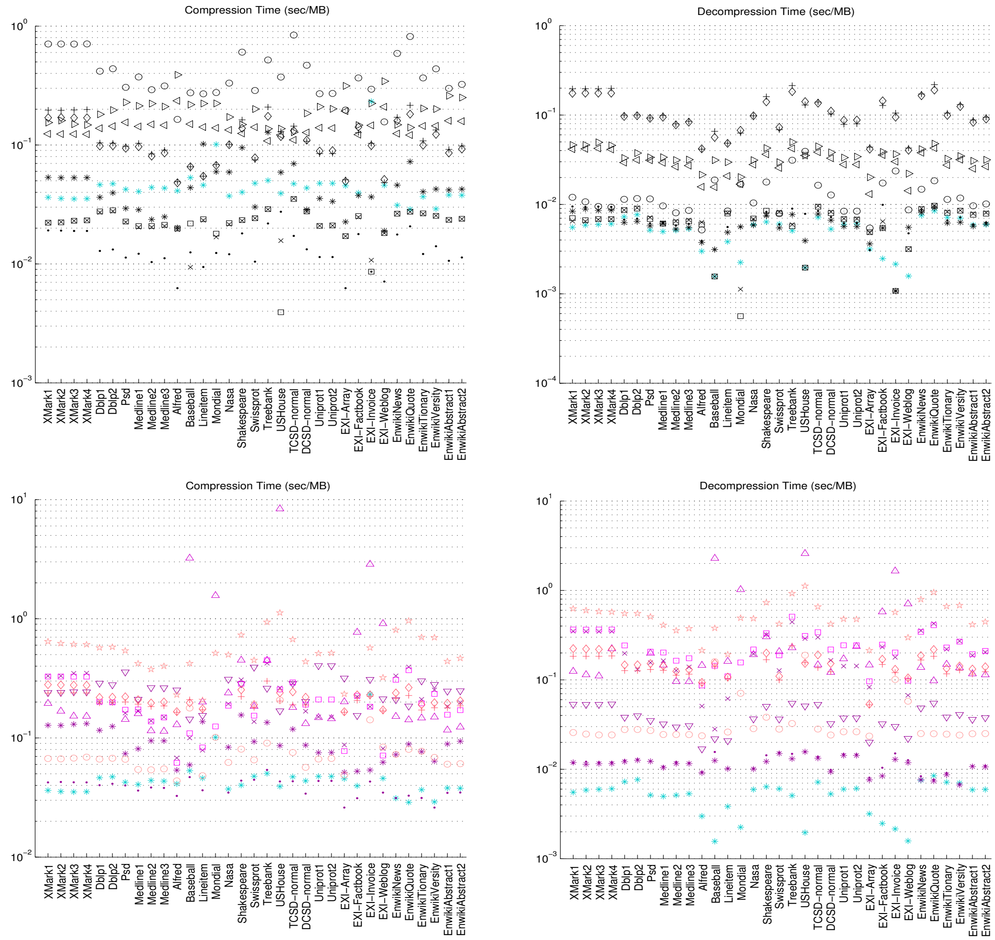


Fig. 11. Compression and decompression times. Comparison of XWT with general text compressors (top) and XML conscious non-queriable compressors (bottom).

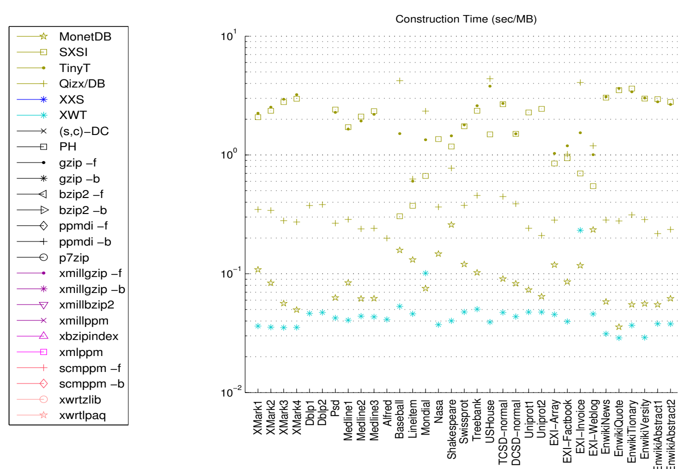


Fig. 12. Construction times of queriable solutions.

Q01: /site/closed_auctions/closed_auction/annotation/description/text/keyword
Q02: //closed_auction//keyword
Q03: /site/closed_auctions/closed_auction//keyword
Q04: /site/closed_auctions/closed_auction[./annotation/description/text/keyword]/date
Q05: /site/closed_auctions/closed_auction[./descendant::keyword]/date
Q06: /site/people/person[./profile/gender and ./profile/age]/name
Q07: /site/people/person[./phone or ./homepage]/name
Q08: /site/people/person[./address and (./phone or ./homepage) and (./creditcard or ./profile)]/name
Q09: /site/regions/*/item[./parent::namerica or ./parent::samerica]/name
Q10: //keyword/ancestor::listitem/text/keyword
Q11: //happiness/ancestor::closed_auction/annotation/author
Q12: /site/open_auctions/open_auction/bidder[./following-sibling::bidder]
Q13: /site/*/person[./homepage/following-sibling::creditcard]/name
Q14: /site/open_auctions/open_auction/bidder[./preceding-sibling::bidder]
Q15: /site/people/person/*/gender[./preceding-sibling::education]
Q16: /site/regions/*/item[./following::item]/name
Q17: /site/open_auctions/open_auction/reserve/following::happiness
Q18: //type/preceding::price
Q19: /site/regions/*/item[./preceding::item]/name
Q20: //person[./profile/@income]/name
Q21: //open_auction[./privacy]/itemref/@item

Q22: //mailbox/*/*/keyword
Q23: //namerica/*/mailbox//*/*/keyword
Q24: //open_auction/*/author
Q25: //regions/*/*/*/*/*/parlist//emph
Q26: //categories/*/description/*/*/keyword
Q27: //categories/*/description//*/*/keyword
Q28: //keyword/parent::*/parent::*/parent::mail/date
Q29: //author/parent::*/parent::open_auction/itemref
Q30: //parlist/parent::*/parent::*/parent::*/parent::*/parent::*/parent::regions
Q31: //keyword/parent::*/parent::*/ancestor::description/parent::category/name
Q32: //keyword/parent::*/ancestor::description/parent::item[./parent::namerica]/location
Q33: //open_auction[.//*/*/@person]/seller
Q34: //person[.//*/*/@category]/homepage
Q35: //person[./*/*/@open_auction]/name
Q36: //categories//*/@id
Q37: //person//*/@income
Q38: /*/*/*//*//*//*/*/*/*
Q39: /*/*/*/*/*/*/*/*/*
Q40: /*//*/*/*/*
Q41: /*/*/*/*
Q42: /*

Fig. 13. Groups of queries `Structural` (left) and `Wildcards` (right).

Q43: //*
Q44: //edge
Q45: //australia
Q46: //province
Q47: //age
Q48: //street
Q49: //homepage
Q50: //parlist
Q51: //keyword
Q52: //date
Q53: //time
Q54: //@*
Q55: //@from
Q56: //@featured
Q57: //@income
Q58: //@id

Q59: //mail//text[contains(.,"image")]
Q60: //item/location[contains(.,"Island")]
Q61: //location[.="Israel"]
Q62: /site/regions/europe/*/location[.="United States"]
Q63: //open_auction/bidder[./date="09/13/1998"]
Q64: //payment[contains(.,"Creditcard")]
Q65: //australia//payment[contains(.,"Personal Check, Cash")]/parent::item/@id
Q66: //namerica//payment[contains(.,"Personal Check, Cash")]
Q67: //text[contains(.,"weaker dove")]
Q68: //annotation[contains(.,"dove miserable")]
Q69: //person/profile/@income[.="9876.00"]
Q70: /site/regions/*/item[./@featured="yes"]/name
Q71: /site//interest[./@category="category266"]
Q72: //interest/@category[.="category328"]
Q73: //@category[.="category328"]

Fig. 14. Groups of queries `Names` (left) and `Text` (right).

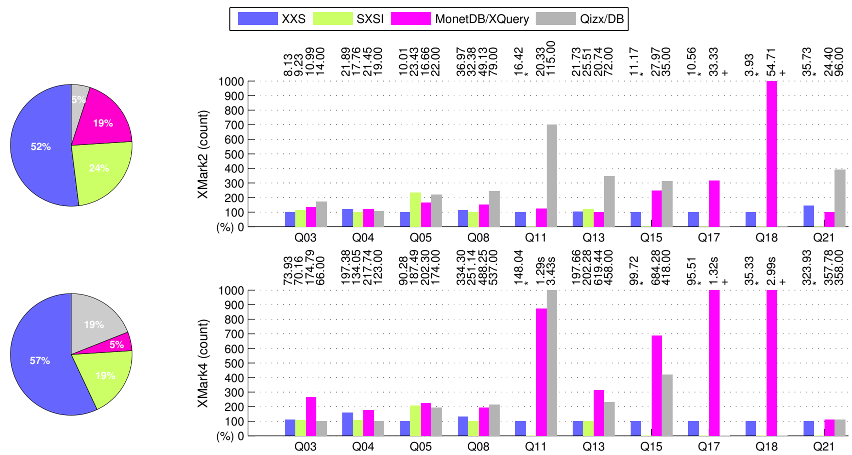


Fig. 15. *Count:* ratio of queries of group `Structural` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).

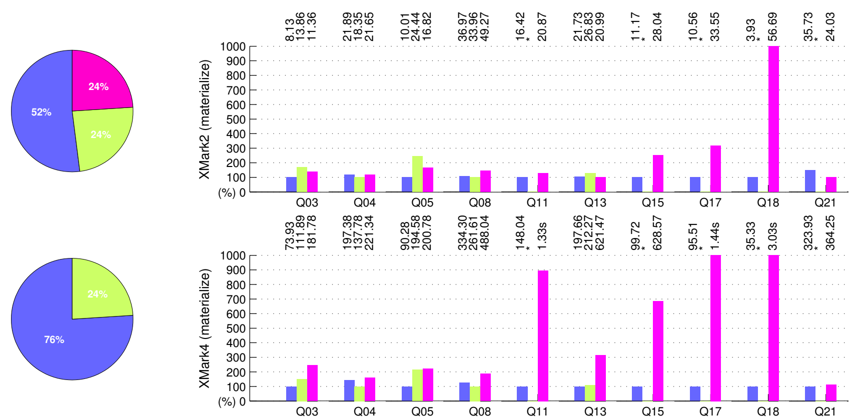


Fig. 16. *Materialize:* ratio of queries of group `Structural` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).

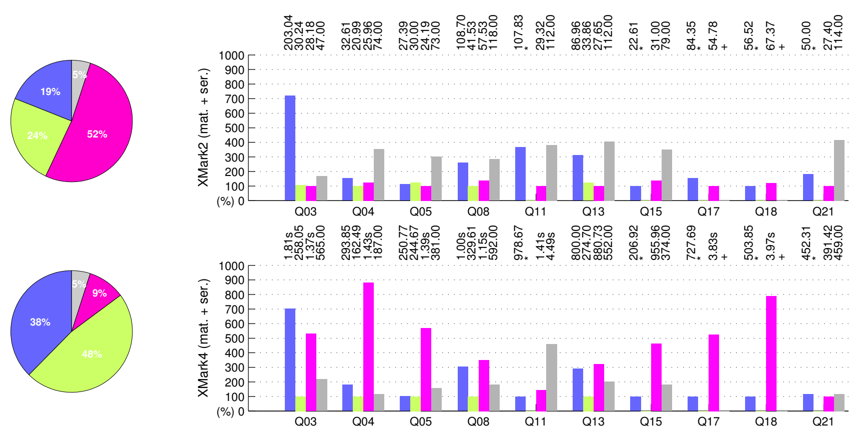


Fig. 17. *Materialize + Serialize:* ratio of queries of group `Structural` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).

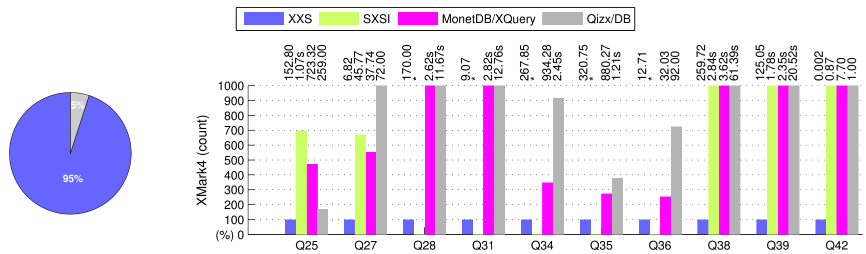


Fig. 18. *Count:* ratio of queries of group `Wildcards` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

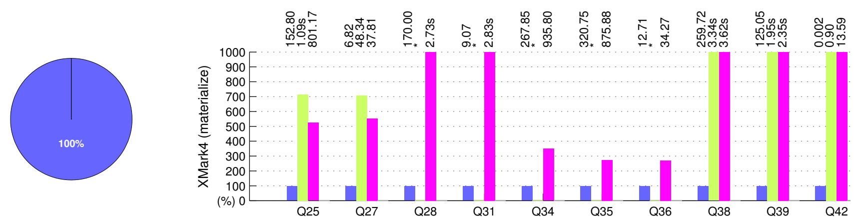


Fig. 19. *Materialize:* ratio of queries of group `Wildcards` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

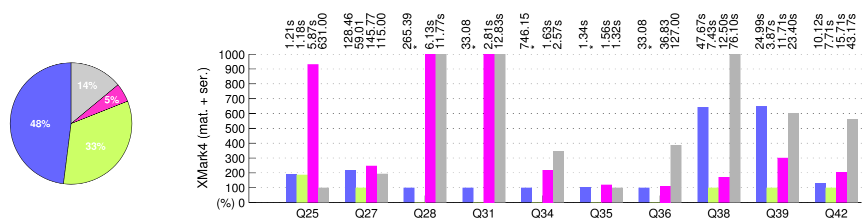


Fig. 20. *Materialize + Serialize:* ratio of queries of group `Wildcards` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

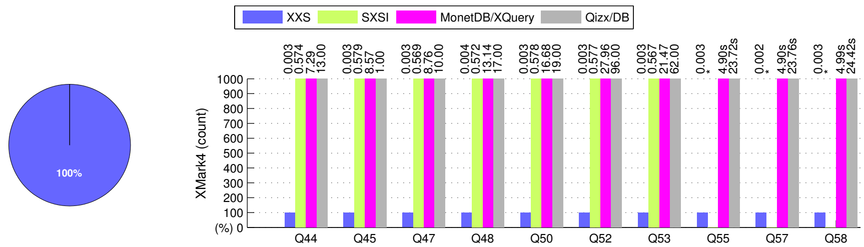


Fig. 21. *Count:* ratio of queries of group `Names` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

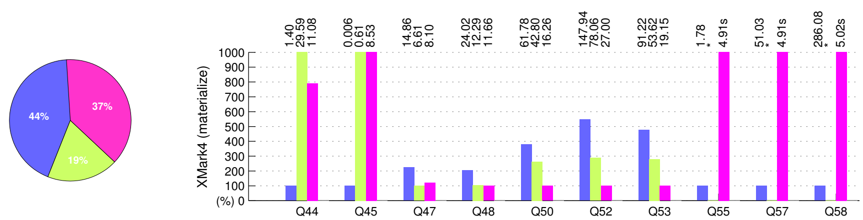


Fig. 22. *Materialize:* ratio of queries of group `Names` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

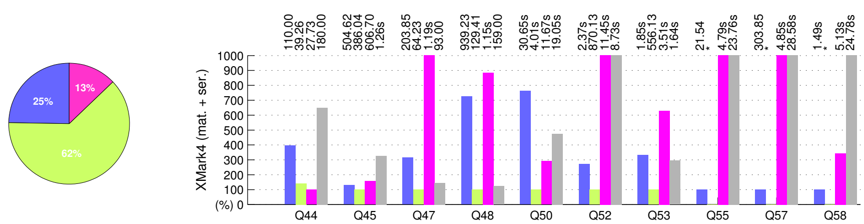


Fig. 23. *Materialize + Serialize:* ratio of queries of group `Names` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark4` (right).

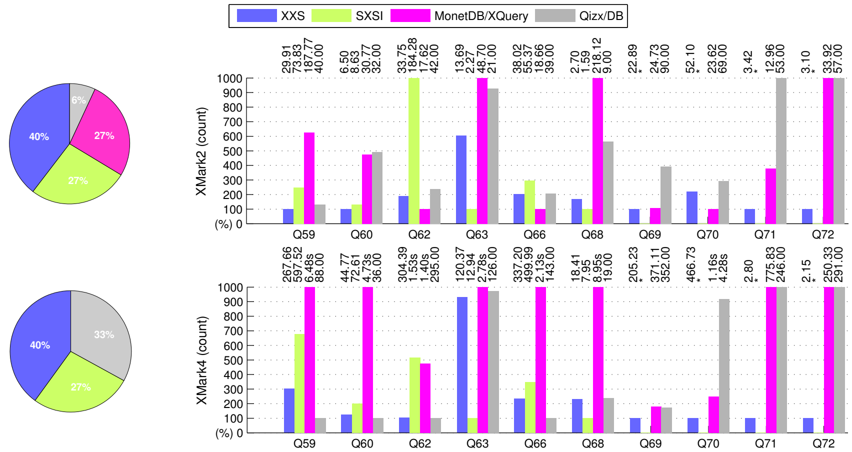


Fig. 24. *Count:* ratio of queries of group `Text` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).

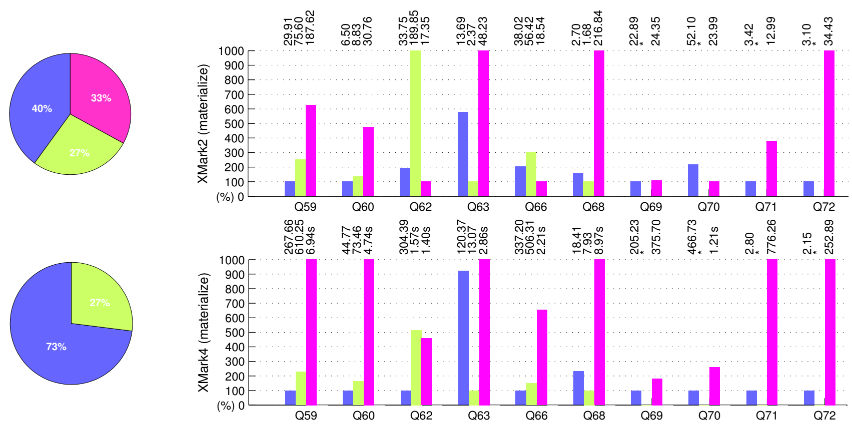


Fig. 25. *Materialize:* ratio of queries of group `Text` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).

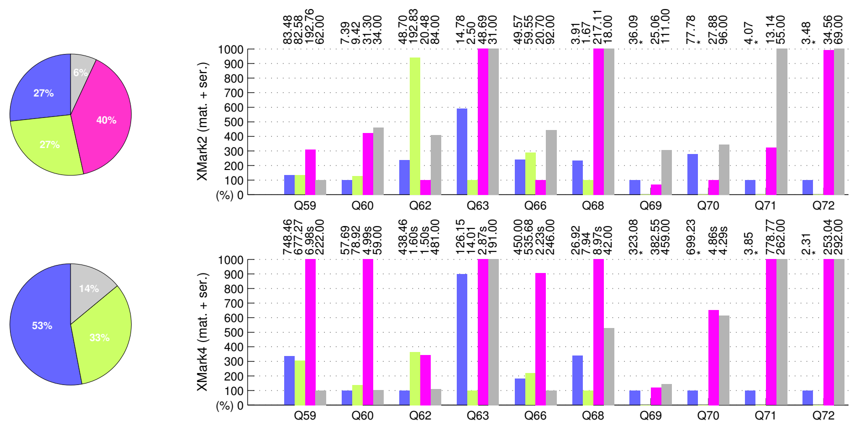


Fig. 26. *Materialize + Serialize:* ratio of queries of group `Text` for which each system obtained the best running times (left). Detailed performance analysis for a selection of queries over `XMark2` and `XMark4` (right).