XQzip: Querying Compressed XML Using Structural Indexing

James Cheng and Wilfred Ng

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{csjames, wilfred}@cs.ust.hk

Abstract. XML makes data flexible in representation and easily portable on the Web but it also substantially inflates data size as a consequence of using tags to describe data. Although many effective XML compressors, such as XMill, have been recently proposed to solve this data inflation problem, they do not address the problem of running queries on compressed XML data. More recently, some compressors have been proposed to query compressed XML data. However, the compression ratio of these compressors is usually worse than that of XMill and that of the generic compressor gzip, while their query performance and the expressive power of the query language they support are inadequate.
In this paper, we propose XQzip, an XML compressor which supports querying compressed XML data by imposing an indexing structure, which we call Structure Index Tree (SIT), on XML data. XQzip addresses both the compression and query performance problems of existing XML compressors. We evaluate XQzip's performance extensively on a wide spectrum of benchmark XML data sources. On average, XQzip is able to achieve a compression ratio 16.7% better and a querying time 12.84 times less than another known queriable XML compressor. In addition, XQzip supports a wide scope of XPath queries such as multiple, deeply nested predicates and aggregation.

1 Introduction

XML has become the de facto standard for data exchange. However, its flexibility and portability are gained at the cost of substantially inflated data, which is a consequence of using repeated tags to describe data. This hinders the use of XML in both data exchange and data archiving. In recent years, many XML compressors have been proposed to solve this data inflation problem. There are two types of compressions: *unqueriable compression* and *queriable compression*.

The unqueriable compression, such as XMill [8], makes use of the similarities between the semantically related XML data to eliminate data redundancy so that a good compression ratio is always guaranteed. However, in this approach the compressed data is not directly usable; a full chunk of data must be first decompressed in order to process the imposed queries.

```
1. <site>
2.  <open_auctions>
3.   <open_auction id="open1">
4.    <initial>$12.00</initial>
5.    <bid>
6.     <date>12/02/2000</date>
7.     <increase>$2.00</increase>
8.    </bid>
9.    <bid>
10.    <date>12/03/2000</date>
11.    <increase>$1.50</increase>
12.   </bid>
13.   <seller person="person71"/>
14.  </open_auction>
15.  <open_auction id="open2">
16.   <initial>$500.00</initial>
17.   <seller person="person8"/>
18.  </open_auction>
19.  <open_auction id="open3">
20.   <initial>$1.50</initial>
21.   <bid>
22.    <date>11/29/2002</date>
23.    <increase>$0.50</increase>
24.   </bid>
25.   <seller person="person15"/>
26.  </open_auction>
27.  <open_auction id="open4">
28.   <initial>$100.00</initial>
29.   <seller person="person11"/>
30.  </open_auction>
31.  <open_auction id="open5">
32.   <initial>$8.50</initial>
33.   <bid>
34.    <date>08/20/2002</date>
35.    <increase>$5.00</increase>
36.   </bid>
37.   <seller person="person7"/>
38.  </open_auction>
39. </open_auctions>
40. </site>
```

Fig. 1. A Sample Auction XML Extract

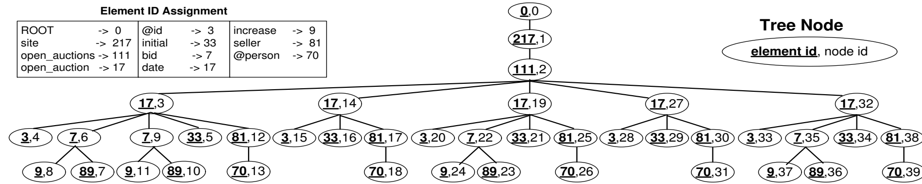

Fig. 2. Structure Tree (contents of the *ext*s not shown) of the Auction XML Extract

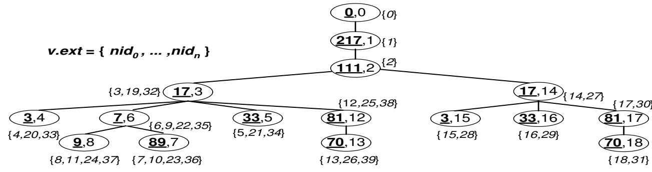

Fig. 3. SIT of the Auction Structure Tree

The queriable compression encodes each of the XML data items individually so that the compressed data item can be accessed directly without a full decompression of the entire file. However, the fine-granularity of the individually compressed data unit does not take advantage of the XML data commonalities and, hence, the compression ratio is usually much degraded with respect to the full-chunked compression strategy used in unqueriable compression.

The queriable compressors, such as XGrind [14] and XPRESS [10], adopts homomorphic transformation to preserve the structure of the XML data so that queries can be evaluated on the structure. However, the preserved structure is always too large (linear in the size of the XML document). It will be very inefficient to search this large structure space, even for simple path queries. For example, to search for bidding items with an initial price under $10 in the compressed file of the sample XML extract shown in Fig. 1, XGrind parses the entire compressed XML document and, for each encoded element/attribute parsed, it has to match its incoming path with the path of the input query. XPRESS makes an improvement as it reduces the element-by-element matching to path-by-path matching by encoding a path as a distinct interval in [0.0,1.0), so that a path can be matched using the containment relationships among the intervals. However, the path-by-path matching is still inefficient since most paths are duplicate in an XML document, especially for those *data-centric* XML documents.

Contributions. We propose XQzip, which has the following desirable features: (1) achieves a good compression ratio and a good compression/decompression time; (2) supports efficient query processing on compressed XML data; and (3) supports an expressive query language. XQzip provides feasible solutions to the problems encountered with the queriable and unqueriable compressions.

Firstly, XQzip removes the duplicate structures in an XML document to improve query performance by using an indexing structure called the *Structure Index Tree* (or *SIT*). An example of a SIT is shown in Fig. 3, which is the index of the tree in Fig. 2, the structure of the the sample XML extract in Fig. 1. Note that the duplicate structures in Fig. 2 are eliminated in the SIT. In fact, large portions of the structure of most XML documents are redundant and can be eliminated. For example, if an XML document contains 1000 repetitions of our sample XML extract (with different data contents), the corresponding tree structure will be 1000 times bigger than the tree in Fig. 2. However, its SIT will essentially have the same structure as the one in Fig. 3, implying that the search space for query evaluation is reduced 1000 times by the index.

Secondly, XQzip avoids full decompression by compressing the data into a sequence of blocks which can be decompressed individually and at the same time allow commonalities of the XML data to be exploited to achieve a good compression. XQzip also effectively reduces the decompression overhead in query evaluation by managing a buffer pool for the decompressed blocks of XML data.

Thirdly, XQzip utilizes the index to query the compressed XML data. XQzip supports a large portion of XPath [15] queries such as multiple and deeply nested predicates with mixed value-based and structure-based query conditions, and aggregations; and it extends an XPath query to select an arbitrary set of distinct elements with a single query. We also give an easy mapping scheme to make the verbose XPath queries more readable. In addition, we devise a simple algorithm to evaluate the XPath [15] queries in polynomial time in the average-case.

Finally, we evaluate the performance of XQzip on a wide variety of benchmark XML data sources and compare the results with XMill, gzip and XGrind for compression and query performance. Our results show that the compression ratio of XQzip is comparable to that of XMill and approximately 16.7% better than that of XGrind. XQzip's compression and decompression speeds are comparable to that of XMill and gzip, but several times faster than that of XGrind. In query evaluation, we record competitive figures. On average, XQzip evaluates queries 12.84 times faster than XGrind with an initially empty buffer pool, and 80 times faster than XGrind with a warm buffer pool. In addition, XQzip supports efficient processing of many complex queries not supported by XGrind. Although we are not able to compare XPRESS directly due to the unavailability of the code, we believe that both our compression and query performance are better than that of XPRESS, since XPRESS only achieves a compression ratio comparable to that of XGrind and a query time 2.83 times better than that of XGrind, according to XPRESS's experimental evaluation results [10].

Related Work. We are also aware of another XML compressor, XQueC [2],

which also supports querying. XQueC compresses each data item individually and this usually results in a degradation in the compression ratio (compared to XMill). An important feature of XQueC is that it supports efficient evaluation of XQuery [16] by using a variety of structure information, such as dataguides [5], structure tree and other indexes. However, these structures, together with the pointers pointing to the individually compressed data items, would incur huge space overhead. Another queriable compression is also proposed recently in [3], which compresses the structure tree of an XML document to allow it to be placed in memory to support Core XPath [6] queries. This use of the compressed structure is similar to the use of the SIT in XQzip, i.e. [3] condenses the tree edges while the SIT indexes the tree nodes. [3] does not compress the textual XML data items and hence it cannot be served as a direct comparison.

This paper is organized as follows. We outline the XQzip architecture in Section 2. Section 3 presents the SIT and its construction algorithm. Section 4 describes a queriable, compressed data storage model. Section 5 discusses query coverage and query evaluation. We evaluate the performance of XQzip in Section 6 and give our concluding remarks and discuss our future work in Section 7.

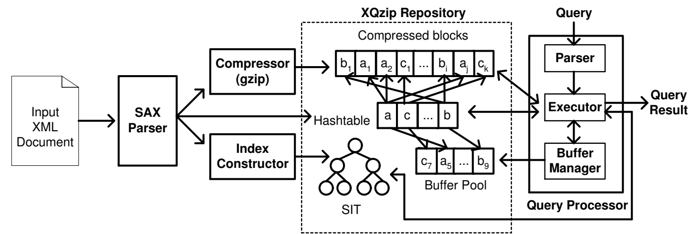

Fig. 4. Architecture of XQzip

2 The Architecture of XQzip

The architecture of XQzip consists of four main modules: the *Compressor*, the *Index Constructor*, the *Query Processor*, and the *Repository*. A simplified diagram of the architecture is shown in Fig. 4. We describe the operations related to the processes of compression and querying.

For the compression process, the input XML document is parsed by the *SAX Parser* which distributes the XML data items (element contents and attribute values) to the Compressor and the XML structure (tags and attributes) to the Index Constructor. The Compressor compresses the data into blocks which can be efficiently accessed from the *Hashtable* where the element/attribute names are stored. The Index Constructor builds the SIT for the XML structure.

For the querying process, the *Query Parser* parses an input query and then the *Query Executor* uses the index to evaluate the query. The Executor checks with the *Buffer Manager*, which applies the LRU rule to manage the *Buffer Pool* for the decompressed data blocks. If the data is already in the Buffer Pool, the

Executor retrieves it directly without decompression. Otherwise, the Executor communicates with the Hashtable to retrieve the data from the compressed file.

3 XML Structure Index Trees (SITs)

In this section we introduce an effective indexing structure called a Structure Index Tree (or a SIT) for XML data. We first define a few basic terminologies used to describe the SIT and then present an algorithm to generate the SIT.

3.1 Basic Notions of XML Structures

We model the structure of an XML document as a tree, which we call the *structure tree*. The structure tree contains only a root node and element nodes. The element nodes represent both elements and attributes. We add the prefix '@' to the attribute names to distinguish them from the elements. We assign a *Hash ID* to each distinct tag/attribute name and store it in a hashtable, i.e. the *Hashtable* in Fig. 4. The XML data items are separated from the structure and are compressed into different blocks accessible via the Hashtable. Hence, no text nodes are considered in our model. We do not model namespaces, PIs and comments for simplicity, though it is a straightforward extension to include them.

Formally, the structure tree of an XML document is an unranked, ordered tree, $T = (V_T, E_T, ROOT)$, where V_T and E_T are the set of tree nodes and edges respectively, and $ROOT$ is the unique root of T. We define a tree node $v \in V_T$ by $v = (eid, nid, ext)$, where $v.eid$ is the Hash ID of the element/attribute being modelled by v; $v.nid$ is the unique node identifer assigned to v according to document order; initially $v.ext = \{v.nid\}$. We represent each node v by the pair $(v.eid, v.nid)$. The pair $(ROOT.eid, ROOT.nid)$ is uniquely assigned as $(0, 0)$. In addition, if a node v has n (ordered) children $(\beta_1, \ldots, \beta_n)$, their order in T is specified as: $v.\beta_1.eid \leq v.\beta_2.eid \leq \cdots \leq v.\beta_n.eid$; and if $v.\beta_i.eid = v.\beta_{i+1}.eid$, then $v.\beta_i.nid < v.\beta_{i+1}.nid$. This node ordering accelerates node matchings in T by an approximate factor of 2, since we match two nodes by their *eid*s and on average, we only need to search half of the children of a given node.

Definition 1. (Branch and Branch Ordering) *A branch of* T*, denoted as* b*, is defined by* $b = v_0 \rightarrow \cdots \rightarrow v_i \rightarrow \cdots \rightarrow v_p$*, where* v_p *is a leaf node in* T *and* v_{i-1} *is parent of* v_i *for* $0 < i \leq p$*. Let* B *be a set of branches of a tree or a subtree. A branch ordering* $\prec$ *on* B *is defined as:* $\forall b_1, b_2 \in B$*, let* $b_1 = u_0 \rightarrow \cdots \rightarrow u_p$ *and* $b_2 = v_0 \rightarrow \cdots \rightarrow v_q$*,* $b_1 \prec b_2$ *implies that there exists some* i *such that* $u_i.nid = v_i.nid$ *and* $u_{i+1}.nid \neq v_{i+1}.nid$*, and either (1)* $u_{i+1}.eid < v_{i+1}.eid$*, or (2)* $u_{i+1}.eid = v_{i+1}.eid$ *and* $u_{i+1}.nid < v_{i+1}.nid$*.*

For example, given b_1 = (0,0) $\rightarrow \cdots \rightarrow$ (3,4), b_2 = (0,0) $\rightarrow \cdots \rightarrow$ (9,11), b_3 = (0,0) $\rightarrow \cdots \rightarrow$ (3,20) in Fig. 2, we have $b_1 \prec b_2$, $b_2 \prec b_3$ and $b_1 \prec b_3$. We can describe a tree as the sequence of all its branches ordered by $\prec$. For example, the subtree rooted at the node (17,27) in Fig. 2 can be represented as: (17,27) $\rightarrow \cdots \rightarrow$ (3,28) $\prec$ (17,27) $\rightarrow \cdots \rightarrow$ (33,29) $\prec$ (17,27) $\rightarrow \cdots \rightarrow$ (70,31), while the

tree in Fig. 3 is represented as: x(3,4) $\prec$ x(9,8) $\prec$ x(89,7) $\prec$ x(33,5) $\prec$ x(70,13) $\prec$ x(3,15) $\prec$ x(33,16) $\prec$ x(70,18), where x denotes (0,0) $\rightarrow \cdots \rightarrow$ for simplicity.

Definition 2. (Sit-Equivalence) *Two branches, $b_1 = u_0 \rightarrow \cdots \rightarrow u_p$ and $b_2 = v_0 \rightarrow \cdots \rightarrow v_q$, are SIT-equivalent if $u_i.eid = v_i.eid$ for $0 \leq i \leq p$ and $p = q$. Two subtrees, $t_1 = b_{10} \prec \cdots \prec b_{1m}$ and $t_2 = b_{20} \prec \cdots \prec b_{2n}$, are SIT-equivalent if $t_1.ROOT$ and $t_2.ROOT$ are siblings and, b_{1i} and b_{2i} are SIT-equivalent for $0 \leq i \leq m$ and $m = n$.*

For example, in Fig. 2, the subtrees rooted at the nodes (17,14) and (17,27) are SIT-equivalent subtrees since every pair of corresponding branches in the two subtrees are SIT-equivalent. The SIT-equivalent subtrees are duplicate structures in XML data and thus we eliminate this redundancy by using a merge operator defined as follows.

Definition 3. (Merge Operator) *A merge operator, $Merge_T$, is defined as: $Merge_T$: $(t_1, t_2) \rightarrow t$, where t_1 and t_2 are SIT-equivalent and $t_1.ROOT.nid < t_2.ROOT.nid$, $t_1 = b_{10} \prec \cdots \prec b_{1n}$ and $t_2 = b_{20} \prec \cdots \prec b_{2n}$, and $b_{1i} = u_0 \rightarrow \cdots \rightarrow u_p$ and $b_{2i} = v_0 \rightarrow \cdots \rightarrow v_p$. For $0 \leq i \leq n$, $Merge_T$ assigns $u_j.ext = u_j.ext \cup v_j.ext$ for $0 \leq j \leq p$, and then deletes b_{2i}.*

Thus, the merge operator merges t_1 and t_2 to produce t, where t is SIT-equivalent to both t_1 and t_2. The effect of the merge operation is that the duplicate SIT-equivalent structure is eliminated. We can remove this redundancy in the structure tree to obtain a much more concise structure representation, the *Structure Index Tree* (*SIT*), by applying $Merge_T$ iteratively on the structure tree until no two SIT-equivalent subtrees are left. For example, the tree in Fig. 3 is the SIT for the structure tree in Fig. 2. Note that all SIT-equivalent subtrees in Fig. 2 are merged into a corresponding SIT-equivalent subtree in the SIT.

A structure tree and its SIT are equivalent, since the structures of the deleted SIT-equivalent subtrees are retained in the SIT. In addition, the deleted nodes are represented by their node identifiers kept in the node *ext*s while the deleted edges can be reconstructed by following the node ordering. Since the SIT is in general much smaller than its structure tree, it allows more efficient node selection than its structure tree.

3.2 SIT Construction

In this section, we present an efficient algorithm to construct the SIT for an XML document. We define four node pointers, *parent*, *previousSibling*, *nextSibling*, and *firstChild*, for each tree node. The pointers tremendously speed up node navigation for both SIT construction and query evaluation. The space incurred for these pointers is usually insignificant since a SIT is often very small.

We linear-scan (by SAX) an input XML document only once to build its SIT and meanwhile we compress the text data (detailed in Section 4). For every SAX start/end-tag event (i.e. the structure information) parsed, we invoke the procedure construct_SIT, shown in Fig. 5. The main idea is to operate on a

```
procedure construct_SIT (SAX-Event)
/*  stack is an array keeping the start/end tag information (either START-TAG or END-TAG);
                    top indicates the stack top;    c is the current node pointer;    count initially is set to 0  */
begin
1.      if (SAX-Event is a start-tag event)                    /* an attribute is also  a start-tag event */
2.         create a new node, u,where u.eid  := hash (SAX-Event) and count := count +1, u.nid  := count;
3.         if (stack [top] = START-TAG)
4.            assign u as the firstchild of c;
5.         else
6.            insert u among the siblings of c according to the SIT node ordering;
7.         top := top + 1;   stack [top] := START-TAG;
8.      else if (SAX-Event is an end-tag event)            /* an end-tag event is also passed after processing an attribute value */
9.         if (subtree (c) is SIT-equivalent to subtree (one of c's preceding siblings, u))        /* check by a parallel DFS */
10.           Merge_T( (subtree (u), subtree (c) );
11.        if (stack [top] = START-TAG)                         /* c has no child and the START-TAG was pushed for c */
12.           if (stack [top – 1] = START-TAG)                  /* c is the first child of its parent */
13.              stack [top] := END-TAG;                        /* finish processing c */
14.           else                                              /* c has preceding sibling(s) (processed) */
15.              top := top – 1;                                /* use the previous END-TAG to indicate c has been processed */
16.        else        /* the END-TAG  indicates c's child processed, stack [top–1] must be START-TAG indicating c not processed */
17.           if (stack [top – 2] = START-TAG)                  /* c is the first child of its parent */
18.              top := top – 1; stack [top] := END-TAG;    /* remove c's child's stack and indicates c has been processed */
19.           else                                              /* c's preceding sibling(s) processed */
20.              top := top – 2;             /* use c's preceding sibling's END-TAG, i.e. stack [top–2], to indicate c has been processed */
21.     c := u;
end
```

Fig. 5. Pseudocode for the SIT Construction Procedure

"base" tree and a constructing tree. A constructing tree is the tree under construction for each *start-tag* parsed and it is a subtree of the "base" tree. When an *end-tag* is parsed, a constructing tree is completed. If this completed subtree is SIT-equivalent to any subtree in the "base" tree, it is merged into its SIT-equivalent subtree; otherwise, it becomes part of the "base" tree. We use a stack to indicate the parent-child or sibling-sibling relationships between the previous and the current XML element to build the tree structure. Lines 11-20 maintain the consistency of the structure information and skip redundant information. Hence, the stack size is always less than twice the height of the SIT.

The time complexity is $O(|V_T|)$ in the average-case and $O(|SIT||V_T|)$ in the worse-case, where $|V_T|$ is the number of tags and attributes in the XML document and $|SIT|$ is the number of nodes in the SIT. $O(|SIT||V_T|)$ is the worst-case complexity because we at most compare and merge $2|SIT|$ nodes for each of the $|V_T|$ nodes parsed. However, in most cases only a constant number of nodes are operated on for each new element parsed, resulting in the $O(|V_T|)$ time. The space required is $|V_T|$ for the node exts and at most $2|SIT|$ for the structure since at all time, both the "base" tree and the constructing tree can be at most as large as the final tree (i.e. the SIT).

SIT and F&B-Index. The SIT shares some similar features with the F&B-Index [1, 7]. The F&B-Index uses bisimulation [7, 12] to partition the data nodes while we use SIT-equivalence to index the structure tree. However, the SIT preserves the node ordering whereas bisimulation preserves no order of the nodes. This node ordering reduces the number of nodes to be matched in query evalu-

ation and in SIT construction by an average factor of 50%. The F&B-Index can be computed in time O($m$log$n$), where m and n are the number of edges and nodes in the original XML data graph, by first adding an inverse edge for every edge and then computing the 1-Index [9] using an algorithm proposed in [11]. However, the memory consumption is too high, since the entire structure of an XML document must be first read into the main memory.

4 A Queriable Storage Model for Compressed XML Data

In this section, we discuss a storage model for the compressed XML data. We seek to balance the full-chunked and the fine-grained storage models so that the compression algorithm is able to exploit the commonalities in the XML data to improve compression (i.e. the full-chunk approach), while allowing efficient retrieval of the compressed data for query evaluation (i.e. the fine-grain approach).

We group XML data items associated with the same tag/attribute name into a same data stream (c.f. this technique is also used in XMill [8]). Each data stream is then compressed separately into a sequence of blocks. These compressed blocks can be decompressed individually and hence full decompression is avoided in query evaluation. The problem is that if a block is small, it does not make good use of data commonalities for a better compression; on the other hand, it will be costly to decompress a block if its size is large. Therefore, it is critical to choose a suitable block size in order to attain both a good compression ratio and efficient retrieval of matching data in the compressed file.

We conduct an experiment (described in Section 6.1) and find that a block size of 1000 data records is feasible for both compression and query evaluation. Hence we use it as the default block size for XQzip. In addition, we set a limit of 2 MBytes to prevent memory exhaustion, since some data records may be long. When either 1000 data records have been parsed into a data stream or the size of a data stream reaches 2 MBytes, we compress the stream using gzip, assign an id to the compressed block and store it on disk, and then resume the process.

The start position of a block in the compressed file is stored in the Element Hashtable. (Note that gzip can decompress a block given its start position and an arbitrary data length.) We also assign an *id* to each block as the value of the maximum node identifier of the nodes whose data is compressed into that block. To retrieve the block which contains the compressed data of a node, we obtain the block position by using the containment relationship of the node's node identifier and the ids of the successive compressed blocks of the node's data stream. The position of the node's data is kept in an array and can be obtained by a binary search on the node identifier (in our case, this only takes log*1000* time since each block has at most 1000 records) and the data length is simply the difference between two successive positions.

A desirable feature of the queriable compressors XGrind [14] and XPRESS [10] is that decompression is avoided since string conditions can be encoded to match with the individually compressed data, while with our storage model (partial) decompression is always needed for the matching of string conditions.

However, this is only true for exact-match and numeric range-match predicates, decompression is still inevitable in XGrind and XPRESS for any other value-based predicates such as string range-match, starts-with and substring matches. To evaluate these predicates, our block model is much more efficient, since decompressing x blocks is far less costly than decompressing the corresponding *1000x* individually compressed data units. More importantly, as we will discuss in Section 5.2, our block model allows the efficient management of a buffer pool which significantly reduces the decompression overhead, while the compressed blocks serve naturally as input buffers to facilitate better disk reads.

5 Querying Compressed XML Using SIT

In this section, we present the queries supported by XQzip and show how they are evaluated on the compressed XML data.

5.1 Query Coverage

Our implementation of XQzip supports most of the core features of XPath 1.0 [15]. We extend XPath to select an arbitrary set of distinct elements by a single query and we also give a mapping to reduce the verbosity of the XPath syntax.

XPath Queries. A query specifies the matching nodes by the location path. A location path consists of a sequence of one or more location steps, each of which has an axis, a node test and zero or more predicates. The axis specifies the relationship between the context node and nodes selected by the location step. XQzip supports eight XPath axes: *ancestor*, *ancestor-or-self*, *attribute*, *child*, *descendant*, *descendant-or-self*, *parent* and *self*. XQzip simplifies the node test by comparing just the *eid*s of the nodes. The predicates use arbitrary expressions, which can in turn be a location path containing more predicates and so on recursively, to further refine the set of nodes selected by the location step.

Apart from the comparison operators (=, ! =, >, <, >= and <=) and string operators (*contains*, i.e. substring, and *starts-with*), XQzip supports a complete set of standard aggregation operators (*count* and *sum*, *average*, *minimum* and *maximum*). XQzip also allows structure-based, value-based, and aggregation predicates to be combined by the logical operators (*not*, *or* and *and*).

XPath Group Queries. An XPath query can only specify one distinct element to be selected at a time. We modify the XPath syntax slightly to make it possible to select an arbitrary set of distinct elements by a single query, which we call an *XPath group query*. We use "(" and ")" to indicate the grouping, and "+" to represent the union of elements in a group. For example, the XPath group query "(//Orderitem[/discount[. >= 20% *and* . <= 50%]/(@id + quantity + price))" selects three elements from "Orderitem" with a "discount" of 20-50%.

Evaluating an XPath group query is much more efficient than evaluating a group of XPath queries, since all location paths inside a group share the same

context node addressed by the location path just preceding the group. For example, given $(l/(l_0 + \cdots + l_n))$, we evaluate l only once for all l_i.

Table 1. Abbreviated Syntax

Full Form	Abbr.	Full Form	Abbr.	Full Form	Abbr.	Full Form	Abbr.	Full Form	Abbr.
self	.	descendant	//	logical-not	**!**	sum()	**$S**	text()	**$T**
child	/	ancestor-or-self	.\	logical-or	|	count()	**$C**	wildcard	*
parent	\	descendant-or-self	/.	logical-and	**&**	max()	**$U**	contains	**?=**
ancestor	\\	root	..	average()	**$A**	min()	**$L**	starts-with	**$=**

Abbreviated Syntax. Although the syntax of XPath is straightforward, it is rather verbose as a query language. We therefore map the XPath axes, together with the functions and operators, to more concise syntactic abbreviations, in addition to those existing standard XPath abbreviated syntax [15]. We show the mapping in Table 1. In order to make parsing easier, our query parser requires that predicates in queries be fully parenthesized. Another difference from XPath's abbreviated syntax is that we use '..' to represent the root whereas XPath uses '..' as the *parent* of the context node, since *parent* is already represented by '\' and we wish to address the root directly in a query.

5.2 Query Evaluation

XQzip evaluates queries in four major phases: (1) query parsing; (2) node selection; (3) data retrieval; and (4) query result output.

Query Parsing. The query parser translates an input query into a stream of events represented as integers, with positive values representing the XML elements (i.e. their Hash IDs) and negative values representing other expressions.

Node Selection. Node selection is critical in query evaluation. A survey [6] shows that contemporary XPath query engines evaluate XPath queries in exponential time. The cause of the exponential time evaluation is that for each location step, a set of nodes of size linear in the size of the document may be selected and each node in this set may in turn select a linear number of nodes for the next location step. Hence, the time complexity is $|D|^{|Q|}$, where $|D|$ is the document size and $|Q|$ the query size. Although [6] proposes a polynomial-time XPath evaluation algorithm, it is not applicable with our setting. We propose a simple algorithm which gives polynomial time complexity in the average case.

Our algorithm basically divides an axis closure into two disjoint areas. We associate each node in the SIT a *visited flag*. A subtree is *visited* if its root's visited flag is set. The union of all *visited subtrees* in an axis closure with respect to a context node forms the *visited_closure*, and the *unvisited_closure* is simply the difference between the axis closure and the visited_closure.

We give the core of our query evaluation algorithm in Fig. 6. The idea is as follows: on evaluating $s_i \dots s_n$ where a_i is the descendant or descendant-or-self

axis (and similarly if a_i is the ancestor or ancestor-or-self axis) w.r.t. a context node u, the subtree rooted at u is set to be visited when the evaluation process finishes $s_i \ldots s_n$ w.r.t. u (regardless of the evaluation result), since the result of $s_i \ldots s_n$ will always be the same for the same context node u. Moreover, an ancestor always includes its descendants and hence we set a node visited when it is included in the result set. Consequently, as the evaluation process goes on, more and more subtrees will be visited and the unvisited_closure becomes smaller and even vanishes. This implies that the nodes selected at each location step are no longer linear at later stages of the query evaluation. Hence, we have the average-case polynomial query evaluation time *in the size of the SIT*.

The worst-case time complexity is still exponential, i.e. $|SIT|^{|Q|}$, since the unvisited_closure has no effect on predicates. Nonetheless, predicate evaluation rarely checks all nodes specified by the predicate's location path but it terminates as soon as one evaluation returns true. More importantly, $|SIT|$ is often orders of magnitude smaller than $|D|$, implying that $|SIT|^{|Q|}$ is much smaller than $|D|^{|Q|}$. The space complexity is O($|SIT|$ + $|V_T|$): O($|SIT|$) since we only hold the SIT in memory and all nodes in the result set are distinct (we do not count the space requirements for the buffer pool and for writing query result) and O($|V_T|$) space is needed to indicate which elements are matched for value-based predicates.

```
procedure evaluate_query ( u, i, Q )
/* u is the context node and i is initially set to 0; Q: s0 . . . si . . . sn, where si = < ai, ti, pij > */
begin
1.     for each node v in unvisited_closure ( ai (u) ) do
2.        if ( ti ( v ) is true and for all j, pij is true for v )
3.           if ( i < n )
4.              evaluate_query ( v, i + 1, Q );
5.           else   /* i = n */
6.              include v in the query result set and set v.visited_flag;
7.           if ( ai = \\ )
8.              set v.visited_flag;
9.     if ( ai = // )
10.       set u.visited_flag;
end
```

Fig. 6. Core Query Evaluation Algorithm

Data Retrieval and Decompression. We have described the retrieval of a compressed block and the retrieval of data from a decompressed block in Section 4. Although the data retrieval cost is not expensive, an element may appear in many places of a query or in a set of queries asked consecutively, resulting in a compressed block being retrieved and decompressed many times. Since we use gzip as our underlying compression tool, we cannot do much to improve the time to decompress a block. Instead, we avoid the scenario that the same block being repeatedly decompressed by introducing a buffer pool.

XQzip applies the LRU rule to manage a buffer pool for holding recently decompressed XML data. The buffer pool is modelled as a doubly-linked list with a head and tail pointer and the buffers do not have a fixed size but are allocated dynamically according to decompressed data size. When a new block

is decompressed, the buffer manager appends it to the tail of the list. When a block is accessed again, the buffer manager takes it out from the list and appends it to the tail. We set a memory limit (default 16 MBytes) to the total size of the buffer pool. When the memory limit is reached, the buffer manager removes the buffers at the head of the list until memory is sufficient to allocate a new buffer.

Each buffer in the pool can be instantly accessed from the Hashtable and is assigned an *id* which is the same as the compressed block id, thus, avoiding decompressing a block again if a buffer with the same id is already in the pool.

The data access patterns of queries asked at a certain time are usually similar according to the principle of locality. Therefore, after some queries have been evaluated and the buffers have been initialized, new blocks tend to be decompressed only occasionally. Our experimental evaluation result shows that the buffer pool significantly reduces the querying time: the average querying time measured with a warm (initialized) buffer pool is 5.14 times less than that with a cold buffer pool. Moreover, restoring the original XML document from the compressed file is also much faster with a warm buffer pool.

Query Result Output. The query processor produces the query result specified by the output expression. XQzip allows the following output expressions: (1) *not specified*: all elements in the result set are returned; (2) location path/*text()*: only text contents of the result elements are returned; (3) location path/*op*: one of the five aggregation operations; and (4) [Q]: returns *true* if Q evaluates to be true, *false* otherwise.

6 Experimental Evaluation

We evaluated the performance of XQzip by an extensive set of experiments. All experiments were run on a Windows XP machine with a Pentium 4, 2.4 GHz and 256 MBytes main memory. We compared our compression performance with XMill, gzip and XGrind, and query performance with XGrind. Since XGrind is not able to compress all the datasets used in our evaluation and simply outputs query results as "found" or "not found", we modified the XGrind source code to make it work for all the datasets we used and write query results to a disk file, as XQzip does. We also made XGrind adapt to our experimental platform.

We first studied the effect of using different sized data blocks on the compression and query performance of XQzip; the aim of this experiment is to choose a feasible default block size for XQzip. We then performed, for each data source, four classes of experiments: (1) the effectiveness of the SIT; (2) compression ratios; (3) compression/decompression time; and (4) query performance. We define the compression ratio as: *Compression Ratio* = (1 − *Compressed file size*/*Original XML file size*) ∗ 100%, and we measure all the time in seconds.

We use eight data sources for our evaluation, which cover a wide range of XML data formats and structures. A description of the datasets is given in [4] due to space limit but we give their characteristics in Table 2, where E_num and A_num refer to the number of elements and attributes in the dataset respectively.

Table 2. XML Data Sources

Data Source	Size (MB)	Depth	Tags/Attrs	E_num	A_num
XMark	111	11	86	1666315	381878
OMIM	24.5	5	22	188052	0
DBLP	148	6	41	3883112	471124
SwissProt	109	5	100	2977031	2189859
Treebank	82	36	252	2437666	1
PSD	683	7	72	21305818	1052770
Shakespeare	7.3	6	23	179072	0
Lineitem	30.8	3	19	1022976	1

6.1 Effect of Using Different Block Size

We carried out a set of experiments to explore the effects of using different data block sizes on compression and query performance. We chose three representative documents: SwissProt (which has no heavy text items), XMark (which has a lot of data and one heavy text item) and OMIM (whose data content is dominated by very heavy texts) for running the experiments.

Compression. For all datasets, compression performance is extremely poor for block sizes less than 2 KBytes and improves linearly with the increase in block size (greater than 2 KBytes), but does not improve much (within 10%) for block sizes beyond 100-150 (SwissProt: ∼150, XMark: ∼130, OMIM: ∼100) KBytes.

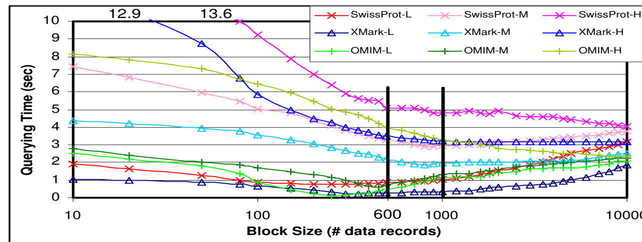

Fig. 7. Querying Time with Different Block Sizes

Query Evaluation. We use range predicates to select a set of queries (the queries are listed in [4] due to the space limit) of different selectivity for each dataset: low-selectivity (appr. 0.01%, 0.03%, 0.05%, 0.08% and 0.1%), medium-selectivity (appr. 0.3%, 0.5%, 0.7%, 1% and 3%) and high-selectivity (appr. 5%, 20%, 50%, 80% and 100%). For each dataset, we plot the average querying time of the queries of each selectivity group, represented by the prefixes L, M and H respectively in Fig. 7. We also found that the block size is actually sensitive to the number of records per block instead of number of bytes per block. We thus measure the block size in terms of number of data records per block.

For all the three data sources, query performance is poor on small block sizes (less than 100 records). High-selectivity queries have better performance on larger block sizes though performance improves only slightly for block sizes

beyond 1000 records. Medium and low selectivity queries have best performance in the range of 500 to 800 records and 250 to 300 records respectively, and their querying time increases linearly for block sizes exceeding the optimal ranges. The difference in querying time of the various selectivity queries with the change in block size is mainly due to the inverse correlation between the decompression time of the different-sized blocks and the total number of blocks to be decompressed w.r.t. a particular block size, i.e. larger blocks have longer decompression time but fewer blocks need be decompressed, and vice versa. Although the optimal block size does not agree for the different data sources and different selectivity queries, we find that within the range of 600 to 1000 data records per block, the querying time of all queries is close to their optimal querying time. We also find that a block size of about 950 data records is the best average.

For most XML documents, a total size of 950 records of a distinct element is usually less than 100 KBytes, a good block size for compression. However, to facilitate query evaluation, we choose a block size of 1000 data records per block (instead of 950 for easier implementation) as the default block size for XQzip, and we demonstrate that it is a feasible choice in the subsequent subsections.

6.2 Effectiveness of the SIT

In this subsection, we show that the SIT is an effective index. In Table 3, $|T|$ represents the total number of tags and attributes in each of the eight datasets, while $|V_T|$ and $|V_I|$ show the number of nodes (presentation tags not indexed) in the structure tree and in the SIT respectively; $|V_I|/|V_T|$ is the percentage of node reduction of the index; Load Time (LT) is the time taken to load the SIT from a disk file to the main memory; and Acceleration Factor (AF) is the rate of acceleration in node selection using the SIT instead of the F&B-Index.

Table 3. Index Size

Data Source	$\|T\|$	$\|V_T\|$	$\|V_I\|$	$\|V_I\| / \|V_T\|$	LT	AF
XMark	2048193	1837608	30071	1.64%	0.67s	2.15
OMIM	188052	188052	445	0.24%	0.07s	2.16
DBLP	4354236	4350639	1877	0.04%	1.62s	2.11
SwissProt	5166890	5166890	1466332	28.38%	5.61s	1.92
Treebank	2437667	2437667	2277202	93.42%	2.26s	1.76
PSD	22358588	22358588	2425868	10.85%	9.97s	2.18
Shakespeare	179072	179072	3514	1.96%	0.07s	2.10
Lineitem	1022977	1022977	19	0.002%	0.42s	1.78

For five out of the eight datasets, the size of the SIT is only an average of 0.7% of the size of their structure tree, which essentially means that the query search space is reduced approximately 140 times. For SwissProt and PSD, although the reduction is smaller, it is still a significant one. The SIT of Treebank is almost the same size as its structure tree, since Treebank is totally irregular and very nested. We remark that there are few XML data sources in real life as irregular as Treebank. Note also that most of the SITs only need a fraction of a second to be

loaded in the main memory. We find that the load time is roughly proportional to $|V_I|/|V_T|$ (i.e. irregularity) and $|V_T|$ of an XML dataset.

We built the F&B-Index (no *idrefs*, presentation tags and text nodes), using a procedure described in [7]. However, it ran out of memory for DBLP, SwissProt and PSD datasets on our experimental platform. Therefore, we performed this experiment on these three datasets on another platform with 1024 MBytes of memory (other settings being the same). On average, the construction (including parsing) of the SIT is 3.11 times faster than that of the F&B-Index. We next measured the time taken to select each distinct element in a dataset using the two indexes. The AF for each dataset was then calculated as the sum of time taken for all node selections of the dataset (e.g. 86 node selections for XMark since it has 86 distinct elements) using the F&B-Index divided by that using the SIT. On average, the AF is 2.02, which means that node selection using the SIT is faster than that using the F&B-Index by a factor of 2.02.

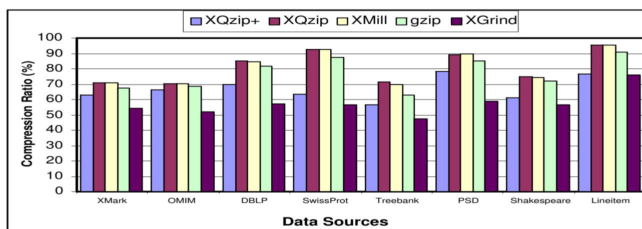

Fig. 8. Compression Ratio

6.3 Compression Ratio

Fig. 8 shows the compression ratios for the different datasets and compressors. Since XQzip also produces an index file (the SIT and data position information), we represent the sum of the size of the index file and that of the compressed file as XQzip+. On average, we record a compression ratio of 66.94% for XQzip+, 81.23% for XQzip, 80.94% for XMill, 76.97% for gzip, and 57.39% for XGrind.

When the index file is not included, XQzip achieves slightly better compression ratio than XMill, since no structure information of the XML data is kept in XQzip's compressed file. Even when the index file is included, XQzip is still able to achieve a compression ratio 16.7% higher than that of XGrind, while the compression ratio of XPRESS only levels with that of XGrind.

6.4 Compression/Decompression Time

Fig. 9a shows the compression time. Since XGrind's time is much greater than that of the others, we represent the time in logarithmic scale for better viewing. The compression time for XQzip is split into three parts: (1) parsing the input XML document; (2) applying gzip to compress data; and (3) building the SIT.

The compression time for XMill is split into two parts as stated in [8]: (1)parsing and (2) applying gzip to compress the data containers. There is no split for gzip and XGrind. On average, XQzip is about 5.33 times faster than XGrind while it is about 1.58 times and 1.85 times slower than XMill and gzip respectively. But we remark that XQzip also produces the SIT, which contributes to a large portion of its total compression time, especially for the less regular data sources such as Treebank.

Fig. 9b shows the decompression time for the eight datasets. The decompression time here refers to the time taken to restore the original XML document. We include the time taken to load the SIT to XQzip's decompression time, represented as XQzip+. On average, XQzip is about 3.4 times faster than XGrind while it is about 1.43 time and 1.79 times slower than XMill and gzip respectively, when the index load time is not included. Even when the load time is included, XQzip's total time is still 3 times shorter than that of XGrind.

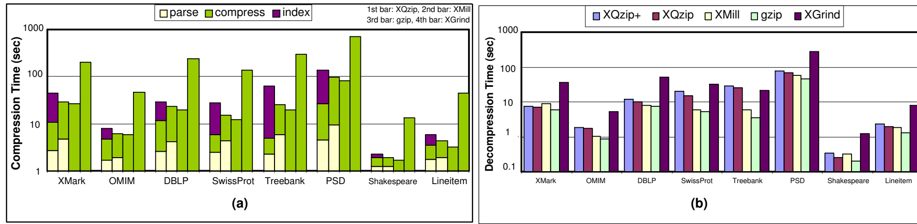

Fig. 9. (a) Compression Time (b) Decompression Time (Seconds in $\log_{10}$ scale)

6.5 Query Performance

We measured XQzip's query performance for six data sources. For each of the data sources, we give five representative queries which are listed in [4] due to the space limit. For each dataset except Treebank, Q1 is a simple path query for which no decompression is needed during node selection. Q2 is similar to Q1 but with an exact-match predicate on the result nodes. Q3 is also similar to Q1 but it uses a range predicate. The predicates are not imposed on intermediate steps of the queries since XGrind cannot evaluate such queries. Q4 and Q5 consists multiple and deeply nested predicates with mixed structure-based, value-based, and aggregation conditions. They are used to evaluate XQzip's performance on complex queries. The five queries of Treebank are used to evaluate XQzip's performance on extremely irregular and deeply nested XML data.

We recorded the query performance results in Table 4. Column (1) records the sum of the time taken to parse the input query and to select the set of result nodes. In case decompression is needed, the time taken to retrieve and decompress the data is given in Column (2). Column (3) and Column (4) give the time taken to write the textual query results (decompression may be needed) and the index of the result nodes respectively. Column (5)is the total querying time,

Table 4. Query Evaluation Results

Data Sources		(1) Node Selecting Time (sec)	(2) Partial Decomp. Time (sec)	(3) Result (text) Processing Time (sec)	(4) Result (index) Processing Time (sec)	(5) Querying Time (sec) (XQzip-)	(6) Querying Time (sec) (XQzip+)	(7) Querying Time (sec) (XGrind)	(8) Query Result (text) (KBytes)	(9) Query Result (index) (KBytes)
XMark (111MB)	Q1	0.001	---	0.911	0.001	0.913	0.122	22.774	263	40
	Q2	0.001	0.920	0.012	0.001	0.934	0.295	23.067	0.8	0.09
	Q3	0.001	3.395	0.014	0.001	3.411	0.349	35.012	1.74	0.22
	Q4	0.003	---	0.551	0.030	0.584	0.118	---	14999	1256
	Q5	0.831	4.534	0.010	0.001	5.376	1.544	---	0.21	0.03
OMIM (24.5MB)	Q1	0.001	---	0.030	0.001	0.032	0.005	3.513	146	23.6
	Q2	0.001	0.021	0.011	0.001	0.034	0.014	4.690	19.1	2.7
	Q3	0.001	0.036	0.057	0.001	0.095	0.067	6.134	66.8	9.45
	Q4	0.005	---	---	---	0.005	0.005	---	---	---
	Q5	0.012	0.020	0.580	0.001	0.613	0.034	---	1666	274
DBLP (148MB)	Q1	0.001	---	0.370	0.010	0.381	0.034	19.582	7219	621
	Q2	0.001	0.330	0.013	0.001	0.345	0.029	26.108	59	6
	Q3	0.033	0.391	8.997	0.120	9.541	1.543	50.344	22940	1853
	Q4	0.001	---	0.000	0.000	0.001	0.001	---	No Match	No Match
	Q5	0.087	1.122	0.260	0.012	1.481	0.642	---	2312	205
Lineitem (30.8MB)	Q1	0.001	---	0.041	0.001	0.043	0.011	2.336	1176	175
	Q2	0.001	0.031	0.011	0.001	0.044	0.012	2.890	130	16
	Q3	0.001	0.058	0.015	0.001	0.075	0.014	3.210	393	54
	Q4	0.001	---	1.594	0.082	1.677	0.342	---	31539	4024
	Q5	0.002	0.030	---	---	0.032	0.007	---	---	---
Shakespeare (7.3MB)	Q1	0.001	---	0.035	0.001	0.037	0.014	1.311	865	89
	Q2	0.001	0.034	0.002	0.001	0.038	0.016	1.620	0.05	0.001
	Q3	0.001	0.032	0.005	0.001	0.039	0.016	2.312	48	2.3
	Q4	0.005	---	---	---	0.005	0.005	---	---	---
	Q5	0.007	0.032	---	---	0.039	0.014	---	---	---
Treebank (82MB)	Q1	0.321	---	3.304	0.120	3.745	0.674	---	21278	5659
	Q2	0.167	---	0.010	0.001	0.178	0.177	---	0.45	0.12
	Q3	0.183	---	1.012	0.064	1.259	0.453	---	785	204
	Q4	0.124	---	6.123	0.282	6.529	1.003	---	24111	6078
	Q5	0.156	---	6.004	0.274	6.434	0.985	---	24111	6078

which is the sum of Column (1) to (4) (note that each query was evaluated with an initially empty buffer pool). Column (6) records the time taken to evaluate the same queries but with the buffer pool initialized by evaluating several queries containing some elements in the query under experiment prior to the evaluation of the query. Column (7) records the time taken by XGrind to evaluate the queries. Note that XGrind can only handle the first three queries of the first five datasets and does not give an index to the result nodes. Finally, we record the disk file size of the query results in Column (8) and (9). Note that for the queries whose output expression is an aggregation operator, the result is printed to the standard output (i.e. C++ stdout) directly and there is no disk write.

Column (1) accounts for the effectiveness of the SIT and the query evaluation algorithm, since it is the time taken for the query processor to process node selection on the SIT. Compared to Column (1), the decompression time shown in Column (2) and (3) is much longer. In fact, decompression would be much more expensive if the buffer pool is not used. Despite of this, XQzip still achieves an average total querying time 12.84 times better than XGrind, while XPRESS is only 2.83 times better than XGrind. When the same queries are evaluated with a warm buffer pool, the total querying time, as shown in Column (6), is reduced 5.14 times and is about 80.64 times shorter than XGrind's querying time.

7 Conclusions and Future Work

We have described XQzip, which supports efficient querying compressed XML data by utilizing an index (the SIT) on the XML structure. We have demonstrated by employing rich experimental evidence that XQzip (1) achieves comparable compression ratios and compression/decompression time with respect to XMill; (2) achieves extremely competitive query performance results on the compressed XML data; and (3) supports a much more expressive query language than its counterpart technologies such as XGrind and XPRESS. We notice that a lattice structure can be defined on the SIT and we are working to formulate a lattice whose elements can be applied to accelerate query evaluation.

Acknowledgements. This work is supported in part by grants HKUST 6185/02E and HKUST 6165/03E from the Research Grant Council of Hong Kong.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. San Francisco, Calif.: Morgan Kaufmann, c2000.
2. A. Arion and et. al. XQueC: Pushing Queries to Compressed XML Data. In *(Demo) Proceedings of VLDB*, 2003.
3. P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *Proceedings of VLDB*, 2003.
4. J. Cheng and W. Ng. XQzip (long version). http://www.cs.ust.hk/∼csjames/
5. R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and Opeimization in Semistructured Databases. In *Proceedings of VLDB*, 1997.
6. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processsing XPath Queries. In *Proceedings of VLDB*, 2002.
7. R. Kaushik, P. Bohannon, J. F. Naughton and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of SIGMOD*, 2002.
8. H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of SIGMOD*, 2000.
9. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of ICDT*, 1999.
10. J. K. Min, M. J. Park, C. W. Chung. XPRESS: A Queriable Compression for XML Data. In *Proceedings of SIGMOD*, 2003.
11. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6): 973-989, Decemember 1987.
12. D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conf.*, LNCS 104, 176-183. Springer-Verlag, Karlsruhe, 1981.
13. A. R. Schmidt and F. Waas and M. L. Kersten and M. J. Carey and I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of VLDB*, 2002.
14. P. M. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor. In *Proceedings of ICDE*, 2002.
15. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, W3C Recommendation 16 November 1999.
16. World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, W3C Working Draft 22 August 2003.