Descriptive Schema Driven XML Storage[1]

Andrey Fomichev, Maxim Grinev, Sergey Kuznetsov

Institute for System Programming of Russia Academy of Sciences,
B. Kommunisticheskaya, 25, Moscow 109004, Russia
{fomichev, grinev, kuzloc}@ispras.ru
http://modis.ispras.ru

Abstract. Unlike traditional relational or object-oriented databases, XML databases require no schema defined in advance. To provide the benefits of a schema in such environments, the notion of descriptive schema (that is also called data guide) was introduced. Descriptive schema is a concise and accurate structural summary of an XML database. It serves as dynamic schema, generated from the database. Descriptive schema helps the user to formulate meaningful queries to the documents without predefined schema. Descriptive schema is also used for query optimization as a basis for query rewriting, query type inference and physical plan construction.

In this paper we go further and use descriptive schema for organizing storage system. Our approach consists in grouping nodes of XML documents in blocks according to their position in the descriptive schema. Thus, descriptive schema plays a role of index structure for path queries that allows us to avoid tree traversal and minimize a number of blocks accessed.

1 Introduction

There is no doubt that XML has already gained ground as a widespread format for information exchange. With significant growth of amounts of XML data being transmitted industry needs storage systems dealing with huge XML documents in efficient way. Particularly, these systems should handle data in secondary storage. This requires solving the problem of data representation that satisfies several requirements. Firstly, data representation should allow efficient execution of regular path expressions such as XPath [1] queries. Secondly, such systems should be able to process data updates as well as queries. And thirdly, they should take into consideration the fact that data representation in secondary memory influences the implementation of high level query languages such as XQuery [2] and XSLT [3] which are usually implemented on the top of XML storage.

The problem of storing and processing XML documents efficiently has been admitted by the database community as a challenge and caused high research activity in this field. Historically, the first wave of research was adopting relational DBMSs for stor-

[1] The work was partially supported by the Russian Foundation for Basic Research (grant 02-07-90300-в).

ing XML. The whole paper is not enough for detailed description of work that has been done, so we can only recommend a summary [6]. But the result of this research consists in principle constraints of pure relational DBMS to handle XML documents efficiently. Actually, XML documents are stored in relational systems either as atomic entities such as BLOBs or being decomposed into relations. The first way of storing cannot guaranty high performance of query evaluation because we need to extract the whole document from database. The second way leads to a great number of resource consuming joins to compose result.

Understanding drawbacks of using relational DBMSs for storing XML caused high activity in development of native XML DBMSs, which would not be straitened by any existing infrastructure. Not pretending to give the complete classification we would like to underline the essential characteristics of these systems. The first group consists of the systems that decompose XML documents at the node level like in case of using relational DBMSs, but make an accent on efficient reconstruction of XML documents (reconstruction is the inverse operation for decomposition). The key to this problem lies in efficient determination of parent-child and ancestor-descendent relationships between nodes. For that reason the notion of *numbering scheme* is introduced. The reconstruction of XML is performed by special join operations (structural joins or containment joins) with the help of the numbering scheme. Usually it is insufficient to have only a numbering scheme and such systems have a set of indexes to get quick access to nodes by name and to avoid tree traversal (because tree traversal leads to a number of structural joins). Most papers, which play around that idea, pay little attention to storage system and updates, but rather concentrate on efficient numbering scheme implementation and optimization of structural joins. More details can be found in [11], [12], [13].

Native XML systems, that make up the second group, work on placement of an XML document (which is essentially a tree) into a number of secondary memory blocks. In this case an XML document is represented as a number of nodes, which are somehow connected with each other by references, and the task is to distribute these nodes among the blocks to satisfy some requirements. For instance, the requirement may consist in minimizing the number of blocks used or in organizing blocks in a balanced tree, so any leaf of the XML tree can be accessed by reading a small fixed number of blocks (usually 2 or 3). A drawback of such approach is that it requires the resource consuming tree traversal operation for path queries, so some indexes should be introduced to speed up query execution. An example of such system which implements this approach is [14].

The third group of native XML DBMSs is the most promising from our point of view. Their main characteristic is that they use descriptive schema or data guide of XML document. Descriptive schema is defined as follows: for each label path in the data, the same path also occurs in the descriptive schema exactly once. Descriptive schema is also referred to as data guide. The earliest work on exploiting descriptive schema for XML data management, as far as we know, is the Lore project [8]. Their data guide was primarily used for query optimization. SphinX [15] system uses descriptive schema for organizing indexes on XML documents. We appreciate these works and think that our approach is closer to theirs than to any other. But they concentrate on indexing XML and do not discuss storage system and updates at all. The last work on compressing XML [16] also takes into account the advantages of descriptive schema. Compressing skeleton that presents the structure part of an XML document they get a variant of data guide, which takes little memory and speeds up query execution. But to the best of our knowledge there is no any native full-featured XML storage system built on the principles of the third group, which not only introduces indexes for XML, but also takes into account how XML is stored in secondary memory and how many I/O operations are performed for queries and updates. The latter is what we do — the approach presented in this paper is used in Sedna native XML DBMS being developed by R&D team MODIS [17]. Also, the ideas described have been approved in BizQuery [18] — a virtual data integration system developed by our team.

The main contributions of this paper are as follows:

- We propose a novel approach to storing XML documents based on descriptive schema. Besides providing efficient support for selection queries, our approach is also suitable for updates of XML documents;
- We propose algorithms for evaluation of important subset of XPath that we call structure path queries;
- We demonstrate feasibility and practical relevance of our approach by a prototype storage system implementation and a number of experiments.

The rest of the paper is organized as follows. In Section 2, we present our data representation for XML and justify our choice. In Section 3, we discuss benefits of our data representation for XPath queries evaluation. Section 4 gives some experimental results. Finally, Section 5 summaries the contribution of this paper and gives outlook to future work.

2 Sedna Storage System

In this section we describe the storage system based on descriptive schema. We start with the explanation of descriptive schema for XML and an example of how we can make it work. Then we present a mechanism of data distribution across secondary memory blocks. We converse about data blocks and node structure organization, numbering scheme and text-enabled nodes. Afterwards we discuss memory management in Sedna DBMS. Subsequently we discuss updates over the presented data structures. We conclude with the comparison of our storage strategy with others.

2.1 Descriptive Schema for XML

Let us consider an example of a simple XML document and its descriptive schema in Fig. 1. By definition, every path of the document has exactly one path in the descriptive schema, and every path of the descriptive schema is a path of the document. Every node of the descriptive schema (we later call it *schema node*) is labeled with type. The type is one of seven types defined in XQuery and XPath Data Model [4]: document node, element node, attribute node, namespace node, processing instruction node, comment node and text node. Some nodes depending on their type also have name. Note that schema nodes of the same type and name are essentially different nodes if they have different ancestors. So, the descriptive schema is a tree.

For most XML documents that you can find in real life the descriptive schema is not very large at all. We assume that one thousand schema nodes is an average gauge for a XML document (which can simply have up to several millions nodes or more). It is hard to find a document with more than 10 thousands of schema nodes. Thus, descriptive schema can easily fit in main memory, which is extremely important for effective query evaluation, as we will see later.

```
<library>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
  </book>
  . . .
  <book>
    <title>An Introduction to Database
           Systems</title>
    <author>Date</author>
    <issue>
      <publisher>Addison-Wesley</publisher>
      <year>2004</year>
    </issue>
  </book>
  <paper>
    <title>A Relational Model for
           Large Shared Data Banks</title>
    <author>Codd</author>
  </paper>
  . . .
  <paper>
    <title>The Complexity of
           Relational Query Languages</title>
    <author>Codd</author>
  </paper>
</library>
```

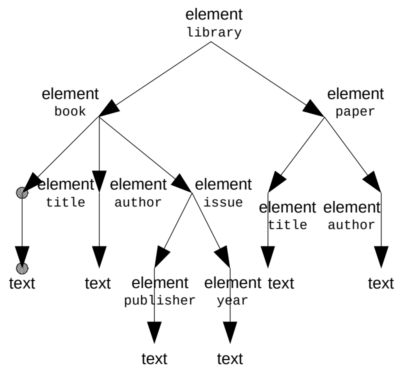

Fig. 1. Sample XML document alone with its descriptive schema

2.1.1 Motivating Example

Let us consider an example of an XPath query to the document in Fig. 1: `/library/book/title`. Having a descriptive schema and this query we will easily find a schema node that satisfies that query. If this schema node had an entry point to the corresponding nodes of the document, then we could simply read them from disk, avoiding traversing XML tree. Hence, descriptive schema plays a role of a naturally built index for path expressions. But having the entry point to the nodes we are looking for, how many blocks will we read from disk and how many I/O operations will be performed? To answer this question we should understand how nodes are stored.

Let us group all the nodes of the XML document in blocks according to the schema nodes and link these nodes by pointers so that we could easily navigate from one node to its sibling, parent and children. So, to evaluate the query discussed we have to read those blocks which belong to element `title` and text schema nodes (marked in

Fig. 1), and only those ones. The blocks to be read contain only nodes that must be presented in the answer and do not contain any other nodes because they belong to other schema nodes and are stored in the other blocks. So we will read only those blocks from disk we need to (remember that traversing descriptive schema does not cost much because it fits in main memory) and minimize the amount of I/O operations.

2.2 Data Organization

To simplify storing of XML documents many storage systems separate the structural part from the textual part of an XML document. In our approach we also separate descriptive schema. Thus, we operate with three entities:

- *Descriptive schema of XML document.* We have already discussed this entity;
- *Structural part of XML document.* Structural part reflects relationships between nodes in XML document. If you consider an arbitrary XML document tree, parent, child, sibling relationships are represented by the structural part;
- *Textual part of XML document.* The text content of nodes of XML document makes up textual part. Those are values of text nodes, attribute nodes and so on.

Descriptive schema is rather small, so we assume that it fits in main memory. It is represented as a number of dynamic structures linked with each other by standard C/C++ pointers. To avoid serialization/deserialization process we use the mechanism of memory-mapped files for storing descriptive schema on disk.

Structural and textual parts are rather large and require lots of disk space. So they are stored in fixed size secondary memory blocks, which can be effectively managed by the buffer manager. The blocks are used for storing node descriptors (which are essentially nodes in terms of XQuery/XPath Data Model) and text. Node descriptors are connected with each other, so having an arbitrary node, system can proceed to its neighbors (siblings, children, parent). We will discuss the details later.

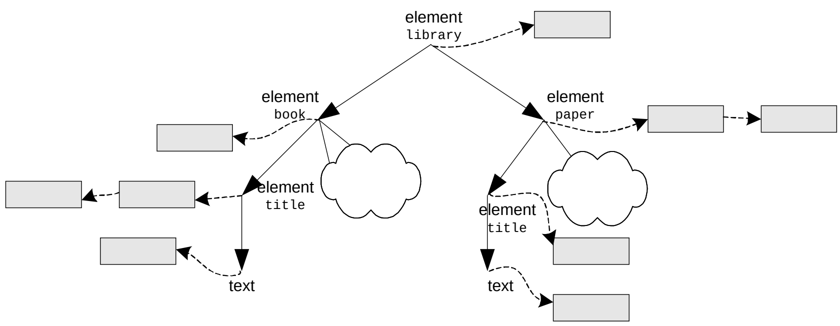

Fig. 2. Data organization in blocks

The descriptive schema serves as an entry point to the structural part. Every schema node has a list of blocks attached which stores node descriptors belonging to this schema node as shown in Fig. 2. For example, element `paper` schema node has a link to a list of two blocks, which store elements with `paper` name and only these elements. So if you want to obtain `/library/paper` nodes you have to find a corresponding path in the descriptive scheme by walking on it and then you get an entry point for the blocks you need. Note that node descriptors may have the same type (element, for example) and name, but if they belong to different schema nodes, they are stored in different block lists (see title elements under the `/library/book` and `/library/paper`).

We have to mention that descriptive schema is a redundant data structure, but it allows us to organize storage by placing node descriptors in corresponding blocks. Thus we need to keep up descriptive schema consistent with data during updates.

2.2.1 Data Blocks and Node Descriptors

In this section we concentrate on the organization of blocks and the structure of node descriptors. Text-enabled nodes will be discussed later.

Data blocks belonging to one schema node are linked via pointers into bidirectional list. Node descriptors in the list are partly ordered according to document order[2]. It means that every node descriptor in the block *i* precedes every node descriptor in the block *j* in document order, if $i < j$ (i.e. the block *i* precedes the block *j* in the list). But node descriptors in the same block are not ordered in document order. This decision has been made to simplify updates and will be discussed later. To reconstruct the order of node descriptors we have introduced special short pointers which are used to link node descriptors from the same block.

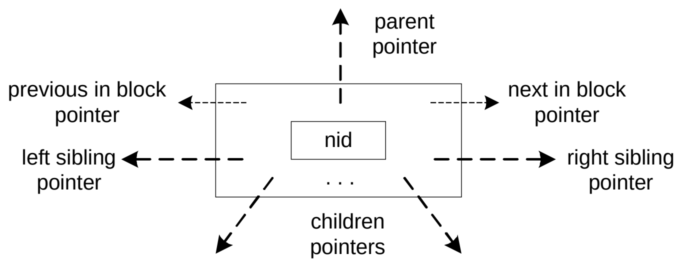

Fig. 3. Node descriptor structure

The common structure of all node descriptors is shown in Fig. 3. Node descriptor consists of the *parent pointer*, the *left sibling pointer* and the *right sibling pointer* (the meaning of which is straightforward) and some other fields. *Next in block* and *previous in block* pointers connect nodes in the same block with the goal to reconstruct document order as was mentioned above. They are short and take only 2 bytes each.

Every node that can have children (i.e. element and document nodes) has variable number of pointers to the children. To save up space we do not store all children pointers in a node, but rather store only a set of pointers to *first children by schema*. Consider an element `library` in Fig. 1; it has several child elements `book` (two

---

[2] Informally, document order is the order returned by an in-order, depth-first traversal of the document [2]

shown) and several child elements paper (two shown as well), but by the descriptive schema library element has only two children. So, exactly two children pointers will be presented in the node descriptor for library element. These are pointers to the first child element with the name book and the first child paper element. Supposing that there should be a large number of books and papers in a library we would save up lots of space. If we would like to select all book elements, which are children of the library element, we have to obtain the first book element and follow 'next in block pointer' to get others (if all children do not fit in one block, we just simply switch to the next block).

Designing the node descriptor structure we wanted to make them fixed-size. It has crucial importance for updates because it simplifies managing of free space in block and insert operations. We did it by introducing *first children by schema* notion as we described above, so even all blocks in the list of blocks have descriptors of the same size. But another problem has arisen — if a new node is inserted in the document for which there is no such schema node, we have to rebuild all blocks in the list belonging to the parent schema node of the inserted node, which is impropriate. So we decided to introduce special attribute ch_num (number of children) to every block. The attribute tells us that all descriptors in this block have exactly this number of children pointers and these pointers are the first ch_num children of the schema node in the corresponding order. As a result, information about children pointers has become a characteristic of a block; so inserting a new node descriptor may lead to the reconstruction of one block only, but not a list of blocks.

The last part of every node descriptor is the nid field, which represents the unique label for a node according to the numbering scheme (see Sect. 2.2.2). Note that node descriptors have no field for its name, because they all have the same name — the name of the corresponding schema node. Every block has a pointer to its schema node, so given an arbitrary node descriptor we can simply look at the header of the block and get a pointer to the schema node, where the name is recorded. This technique allows us to save up space not storing names within node descriptors.

In addition to the fields described above, every node descriptor has its own fields depending on the schema node it belongs to. For example, element node descriptors have a type according to XML Schema [5], text node descriptors have the pointer to a string and the size of the string.

To finish up with node descriptors, we have to mention that all 'long' pointers (siblings, children) are straightforward pointers (C/C++ like) to some objects in our own virtual address space, which allows fast navigation from one node to another without converting these pointers using some tables (there is only one exception — parent pointer, which uses one level of indirection; we will discuss reasons for that in Sect. 2.4). The size of the pointer is 64 bits, so it allows us to address enormous number of objects and operate with really huge databases. Memory management in Sedna DBMS is a topic for another discussion, but we briefly outline it in Sect. 2.3.

2.2.2 Numbering Scheme

The main goal of using numbering scheme is to quickly determine ancestor-descendant relationship between any pair of nodes in the hierarchy of XML data [11].

It can also be used for determining document order relationship. Let us consider a query `/library/*/*/year`. It finds all year elements that are included in library elements. Once all library elements and year elements are found, those two element sets can be joined to produce the answer. This join operation can be performed quickly without tree traversal with the help of the numbering scheme.

This idea was proposed in XISS [11] and a numbering scheme was developed for this DBMS. We appreciate this work, but a drawback of their implementation is that a set of update operations may lead to a reconstruction of full XML tree. They have used a pair of integers as a label for node and make an effort to reserve a diapason of integers for future updates. When the diapason becomes exhausted the XML tree have to be reconstructed.

We have elaborated the idea of a numbering scheme and created our own implementation based on strings with the goal to get rid of tree reconstruction. The idea is on the surface: if we have two strings `str1` and `str2` and `str1 < str2` (lexicographically) then there exists string `str` for which the statement `st1 < str < str2` is true (for example, `(str1= 'abn', str2 = 'ghn') => (str = 'bcb'); (str1 = 'ab', str2 = 'ac') => (str = 'abd')`). In our implementation every node descriptor has a label `nid = (id, d)`. `id` is a string that presents *numbering label* and `d` is a character that presents *delimiter*. String interval `(id, id+d)`, where operation `+` means concatenation of strings, sets the range of numbering labels for all ancestors of the given node. Thus, to check if node 1 with `nid1 = (id1, d1)` is an ancestor to node 2 with `nid2 = (id2, d2)`, we have to test the condition `id1 < id2 < id1+d1`. If it is true, then node 1 is an ancestor for node 2. And for every two nodes `nid1 = (id1, d1)` and `nid2 = (id2, d2)` the following statement is true: `id1 < id2` (lexicographically) if and only if the fist node precedes the second node in document order.

Because of the lack of space we do not present here an algorithm for the insert operation, i.e. how to find a numbering label and delimiter for inserted node. A reader can easily depicture this algorithm by himself. We only say a few words about how we store numbering labels that are variable-length size. If the length of a numbering label is not larger than 8 bytes, we store it inside node descriptor (`nid` field), else we store it outside of node descriptor and `nid` servers as a pointer to that string. We manage 'outside' `nid`s the same way as we manage string data.

2.2.3 Text-enabled Nodes

Text-enabled nodes are those which have a variable-length size data. They are text nodes, attribute nodes and the ones alike. Furthermore, `nid`s may exceed the length of 8 bytes and then they are stored in the same way as variable-length data. We use well-known slotted-page structure method [19] with a little modification. To manage free space in a block effectively we added priority queue, so now it functions similar to malloc implementations.

2.3 Memory Management

The data representation is based on the fact that an XML document is made of a number of nodes somehow distributed across the blocks and these nodes are linked with each other by pointers. So, almost every transfer from one node to another requires dereferencing. Thereafter, optimizing the dereference operation seems to be a primary goal to increase the performance of the system when data being processed fits into main memory. This statement highly correlates with the proposition that a storage system should provide the ground for effective implementation of high-level query languages such as XQuery and XSLT. Queries expressed in these languages require intensive work with stored data structures during joins, transformations, etc.

All these problems were quite actual for object oriented DBMSs, because navigation through object hierarchy was an often operation. Developers have made a great effort studying the problem of management of pointers when blocks are moved between main and secondary memory. The process of transformation of a pointer in secondary memory to the pointer that can be used directly in main memory is called *pointer swizzling* [20]. We do not have enough space to give an overview of pointer swizzling strategies, but we would like to mention that all of them except one (as far as we know) have the following drawback: before you follow the pointer you have to check if it was swizzled or not. The work 'Pointer Swizzling at Page Fault Time' [21] avoids this problem: pointers stored in blocks are real pointers in virtual address space of a process, so when you want to dereference some pointer, you simply follow it. The problem is that the block you want to switch to may not be in memory. In this case you get the memory exception that can be handled and the needed block can be loaded into main memory. After that the query is processed in normal mode.

We use this idea for Sedna memory management with a modification, which consists in extending the size of virtual address space. Standard 32-bit architectures that are widely used nowadays allow addressing only 4Gb of data and operating systems usually restrict its size even more (to 2Gb in Windows 2000 Professional/Server, for example). Thus, we made an effort to extend the size of virtual address space by managing our own layered virtual address space.

The idea of layered virtual address space (LVAS) is to divide the whole huge logical address space, which is addressed by 64-bit pointers on layers. All layers have the same size (we have chosen 1Gb) and are mapped on the same part of process' virtual address space (PVAS) provided by OS. The first 32 bits of a 64-bit pointer identify the layer of the object the pointer references to. Another 32 bits are served to point out the object in this layer. Because data is stored on disk in a number of blocks, we use block as the unit of interaction with disk. The access to blocks of LVAS is provided by mapping addresses of LVAS to addresses of PVAS. Note that we map not layer to layer, but addresses from different layers to addresses of PVAS. So we could work with parts of different layers simultaneously without remapping overhead. The mapping is quite simple: every block addressed by 64-bit pointer (layer, addr) is mapped to process pointer addressed by addr (addrs are equal). This allows us not to store any auxiliary tables to perform mapping. In this case dereferencing is performed by the following algorithm: 1) we dereference the second part of the pointer (addr) like in

C/C++ language; 2) we check that the block addressed by addr has the same layer as the pointer. If any error happens the block is not in memory and we have to read it.

To sum up with the memory management in Sedna DBMS, we would like to emphasize the main goals achieved.

- We emulate 64-bit virtual address space on the standard 32-bit hardware (the platform is Windows 2000);
- Overhead for dereference operation is not much more that for dereferencing standard C/C++ pointer and is caused by support for 64-bit virtual address space;
- And the main result is that we fully avoid swizzling, because data we work with have the same representation in secondary memory and in main memory (because pointers stored in blocks are real pointers in our virtual address space).

2.4 Updates

Developing system, which we believe is highly adequate for selection, we were keeping in mind that it should handle updates as well. By this statement we mean that a local update operation should not cause the global reconstruction of XML tree. In other words, we should be able to estimate the number of blocks accessed by an update operation before it started. And this number should be $O(n)$, where n is a number of nodes modified. Because of the absence of any widely accepted standard for XML updates we speak in terms of micro-operations (e.g. insert/delete node), which can be later used to express updates of any complexity.

Node descriptors are inserted into and are deleted from blocks, which are standard procedures and are not interesting for discussion. One exception is that sometimes block splits. When a node descriptor is inserted into the full block we have to construct two blocks from the existing one to preserve the proper data ordering.

The main question is as follows. What must be modified to evaluate an update operation? To satisfy requirements update operation should change only a neighborhood of the node in scope and avoid ‘mass’ updates. We demonstrate it on the micro-operation *move* (when the block splits, some nodes are moved to another block). For a node being moved we have to modify its left and right sibling and his parent if the given node is the first child of its parent by schema. And at last, all his children have to be modified too, because they all refer to his parent. The number of children can be enormous which leads to a ‘mass’ update. For that reason we have introduced indirection table for nodes, so all parent pointers have one level of indirection. As a result, instead of modifying all children we simply correct record in the indirection table.

To sum up with updates we enumerate below the decisions made to improve their performance.

- node descriptors have a fixed size;
- node-descriptors are partly ordered;
- a numbering scheme does not need tree reconstruction during updates;
- indirection table for parent pointers.

2.5 Comparison with Other Storage Strategies

In conclusion to the description of the Sedna storage system we would like to discuss the advantages and disadvantages of our data structures comparing with other approaches. Obviously, we avoided resource-consuming joins peculiar to the approach based on using relational DBMSs for storing XML. So we concentrate on comparison with native XML DBMSs.

Comparing with the approach, which decomposes document at the node level and uses the numbering scheme for reconstruction, we have to mention that we are ready for using its set of algorithms. These algorithms are based on the notion of numbering scheme, and we support numbering scheme in full measure. We even implemented our own algorithm of managing numbering scheme, which is free from global reordering. Furthermore, we can perform navigational operations more quickly, because for every node we have a set of pointers to neighbors in the XML hierarchy. But the tradeoff is that the size of our data structures is larger than the size of theirs.

Comparing with the approach, which manages XML document as a tree, we avoided the tree traversing operation, which we believe is the main drawback. With the help of descriptive schema we can position to the right place and get access to queried data immediately. Moreover this data is concentrated in several blocks (we know where they are) to speed up processing.

The disadvantage of our approach is that it takes a time for outputting large parts of XML documents, because data is dispersed among a number of blocks and we have to access the same block several times to reconstruct XML tree. But our study shows that for most real life queries the needed blocks remain in buffers and access to it performed almost for free because of using straightforward pointers.

3 Query Evaluation

In this section we demonstrate different execution strategies for regular path expressions which are allowed by our data structures. We concentrate on usage of descriptive schema for answering queries. We assume that data and its descriptive schema have the form that is presented in Fig. 1. Sample queries are shown below.

Sample queries to demonstrate query evaluation strategies

```
Q1: /library/book/issue
Q2: /library/*/title
Q3: //title
Q4: /library/book/[issue/year=2004]/title
Q5: /library/*[.//publisher]
Q6: /*/book[author=”Date”]/issue[year=2004]/publisher
```

The first three queries we call *structure path queries*, because we do not need to make any tests depending on data. In other words, all data, which are read by these queries are used to form the answer. Structure path queries are ideal to be evaluated using descriptive schema. Consider query Q1. We start its evaluation by traversing descriptive schema for the context document and make two transitions from `library` element to `book` element and then to `issue` element. The schema node which is obtained has a pointer to a list of blocks with the data we are looking for. Now we pass through the list of blocks and output a result.

The queries Q2 and Q3 are a bit harder to evaluate. Traversing the descriptive schema we find two schema nodes that satisfy the queries (for Q3 we have to traverse through the whole descriptive schema, but for Q2 only a part). To output the result we cannot just pass through the first list of blocks and then through the second one, because we may break the document order (there may exist a title of a paper which occurs before some book titles in the document). Thus, if the result of schema traversal is several schema nodes, we have to reconstruct document order. The process is performed by merge operation. The merge operation receives several lists of blocks as an input and produces the sequence of node descriptors, which are ordered with respect to document order. The numbering scheme is used for performing this operation. Because node descriptors in the given lists are partly ordered the process of merging is not resource consuming: $O(\sum_i n_i)$ comparisons of `nid`s, where n_i is the number of node descriptors in the i-th list of blocks. Note that as for the query Q1 we read only those blocks we need to (there exists only one difference: for the queries Q2 and Q3 we use `nid`s and it may lead us to reading blocks with `nid`s if they are large and do not fit in node descriptors).

To sum up, the algorithm for evaluation structure path queries consists of two steps. At first we traverse descriptive schema to find the schema nodes which satisfy the query, and at second we pass through the lists of blocks to form a result. If traversing of schema leads to several schema nodes we perform the merge operation.

The last three queries require more effort to be evaluated. Consider query Q4. As for the previous queries we can select `/library/book` elements using the descriptive schema and then apply predicate and the last part of the query using pointers in data. But it seems to us the following algorithm is more attractive. Firstly, we evaluate the structure path query `/library/book/issue/year/text()`. Secondly, we apply the predicate (we select only those nodes, for which text is equal to 2004). And at last, we perform `../../../title` on the result of the previous step. The idea is that we select blocks to which the predicate applies on the first step omitting blocks with book elements. Then we apply the predicate, which we hope cut off lots of data, and only then go up the XML hierarchy to obtain the final result.

The mission of the query Q5 is to show how information about a predicate can be extracted and used to optimize the number of blocks accessed. Executing the `/library/*` query on the first step we get two schema nodes (element `book` and element `paper`), but only the one (element `book`) has descendant element with the name `publisher`. In that way not reading any data block we already can exclude one list of blocks from the answer, because there is no any node descriptor in that list which has the element `publisher` as a descendant.

And at last, the query Q6 demonstrates the importance of the numbering scheme for evaluating a subset of XPath queries. We propose the following strategy for evaluation of this query. Firstly, we evaluate `/library/book/[author=”Date”]` and

/library/book/issue[year=2004] queries as was shown above for the query Q4. On the second step we filter the obtained elements issue by determining ancestor-descendant relationship between them and the selected book elements. We use the numbering scheme for this purpose as was shown in Sect. 2.2.2. The operation which determines ancestor-descendant relationship is similar to merge operation and has the same computational complexity.

Note that for the last three queries we have shown some interesting and not obvious strategies for query evaluation, which are allowed by our data structures. But we do not declare that they are best in all cases. We just reveal them, but the query optimizer should make the final decision which strategy is the best one. It is one of the directions of our future work.

4 Performance Study

Presented in this section are the results of Sedna performance measurement. For data generation we used the XMark benchmark [22]. It allows generating arbitrary amounts of XML data which satisfy the fixed schema (they use DTD auctions). Thus, this benchmark is good for testing scalability of our data structures.

Q1: `document("auctions")/site/people/person/emailaddress`
Q2: `document("auctions")/site/categories/category/description//listitem/text`
Q3: `document("auctions")/site/people/person[profile/@income > 100000]`
Q4: `document("auctions")/site/*/*/seller[@person="person1111"]/..`

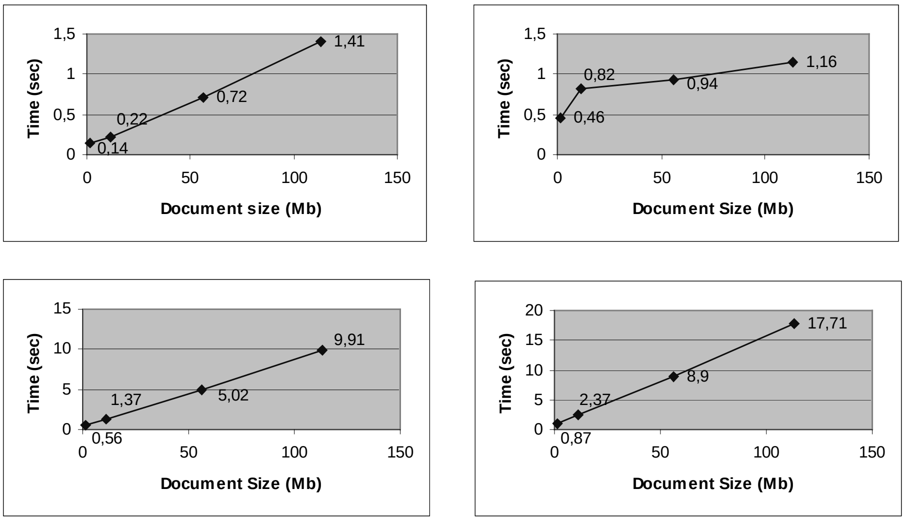

Fig. 4. Sample queries and performance results

The XMark benchmark has a set of predefined queries, but these queries are formulated in XQuery and cover not only path expressions, but other XQuery operations as well. So, we have developed our own set of queries. The four typical queries of this set are shown in Fig. 4.

The environment for performance measurements was the following. The Sedna DBMS was running on the computer with Pentium-IV 1500Mhz and 512Mb RAM. The platform was Windows 2000 Server. Every query was executed from the cold start of the system (buffers were empty). The size of XML file (document auction.xml produced by the generator delivered with XMark) was 1,1Mb, 11,3Mb, 56,2Mb and 113,1Mb. The results of performance measurements are shown in Fig. 4. The size of the produced results is shown in the Table 1.

Table 1. Size of the results for sample queries

Query	Results size (Mb)			
	For 1,1Mb data	For 11,3Mb data	For 56,2Mb data	For 113,1Mb data
Q1	0,017	0,2	0,8	1,6
Q2	0,009	0,1	0,4	1,4
Q3	0,004	0,02	0,1	0,2
Q4	0,003	0,005	0,04	0,2

Note that for all queries the time of query evaluation growths a bit slowly than the size of the data. That means that our data structures can be used for dealing with large volumes of data.

Note that we achieved good performance results without using value indexes. For the last two queries we used sequential scanning. So, introducing value indexes to our system we can significantly improve performance results in the future.

5 Conclusion and Future Work

In the paper we have presented an idea of how XML storage can be organized taking into account a descriptive schema of a document. We also discussed the core techniques of our native XML DBMS Sedna which strongly correlate with this idea. Data structures presented in the paper have a general-purpose nature and can be extended in the future. We are planning to add value-indexes and develop fine grain locking mechanism (we suppose that descriptive schema will be used thoroughly). And the last but not the least, descriptive schema and our data organization plus statistics give ground for query optimization.

And to sum up, we would like to mention that understanding all drawbacks of methods presented in this paper we believe that our way of dealing with XML data is a good alternative for existing ones.

References

1. XML Path Language (XPath) 2.0, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xpath20-20031112/
2. XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xquery-20031112/
3. XSL Transformations (XSLT) Version 2.0, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xslt20-20031112/
4. XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/
5. XML Schema Part 1: Structures, XML Schema Part 2: Datatypes, W3C Recommendation, 02 May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/
6. Tian, F., DeWit, D., Chen, J., Zhang, C.: The Design and Performance Evaluation of Alternative XML Storage Strategies. SIGMOD Record 31(1): 5-10 (2002)
7. Florescu, D., Kossman, D.: A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, Technical Report 3680, INRIA, France, 1999
8. McHugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3):54-66, September 1997
9. Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, Proceedings of the 23th VLDB Conference, Greece, 1997
10. McHugh, J. and Widom J.: Query Optimization for XML, Proceedings of the 25th VLDB Conference, Scotland, 1999
11. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions, Proceedings of the 27th VLDB Conference, Roma, Italy, 2001
12. Jagadish, H., Al-Khalifa, S., Chapman, A., Lakshmanan, L., Nierman, A., Paparizos S., Patel, J., Srivastava D., Wiwatwattana N., Wu, Y. and Yu, C.: TIMBER: A Native XML Database, The VLDB Journal, Volume 11, Issue 4 (2002) pp 274-291
13. Al-Khalifa, S., Jagadish, H., Patel, J., Wu, Y., Koudas, N., Srivastava, D.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching, Proceedings of ICDE 2002, San Jose, California
14. Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system, The VLDB Journal, Volume 11, Issue 4 (2002) pp 292 - 314
15. Leela, K., Haritsa, J.: SphinX: Schema-conscious XML Indexing, Technical Report, TR-2001-04, DSL/SERC, http://dsl.serc.iisc.ernet.in/pub/TR/TR-2001-04.pdf
16. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML, Proceedings of the 29th VLDB Conference, Germany, 2003
17. Management of Data and Information Systems (MODIS) team, http://modis.ispras.ru
18. Antipin, K., Fomichev, A., Grinev, M., Kuznetsov, S., Novak, L., Pleshachkov, P., Rekouts M. and Shiryaev, D.: Efficient Virtual Data Integration Based on XML, Proceedings of ADBIS 2003
19. Silberschatz, A., Korth, H., Sudarshan, S.: Database System Concepts, Third Edition, McGraw-Hill, 1997
20. Garcia-Molina, H., Ullman, J., Widom, J.: Database Systems: The Complete Book, Prentice Hall, 2002
21. Wilson, P., Kakkad, S.: Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware, Proceedings of Workshop on Object Orientation and Operating Systems, Paris, France, 1992
22. XMark — An XML Benchmark Project, http://www.xml-benchmark.org