Sedna: A Native XML DBMS

Maxim Grinev
Institute for System Programming of the Russian Academy of Sciences
B. Kommunisticheskaya, 25, Moscow 109004, Russia
maxim@grinev.net

Andrey Fomichev
Institute for System Programming of the Russian Academy of Sciences
B. Kommunisticheskaya, 25, Moscow 109004, Russia
fomichev@ispras.ru

Sergey Kuznetsov
Institute for System Programming of the Russian Academy of Sciences
B. Kommunisticheskaya, 25, Moscow 109004, Russia
kuzloc@ispras.ru

ABSTRACT

Sedna is an XML database system being developed by the MODIS team at the Institute for System Programming of the Russian Academy of Sciences. Sedna implements XQuery and its data model exploiting techniques developed specially for this language. This paper describes the main choices made in the design of Sedna, sketches its most advanced techniques, and presents its overall architecture. In this paper we primarily focus on physical aspects of the Sedna implementation.

1. INTRODUCTION

Although XQuery is already a powerful and mature language for XML data querying and transformation, it is still a growing language. The authors of XQuery point out two principal directions of the XQuery evolution [4]: (1) extending it with data update facilities, and (2) growing it to a programming language. Thus, future XQuery is going to be a language for querying, updating and general-purpose programming. Implementing XQuery, researchers have been often focused on some of these aspects from the logical viewpoint[1] while an advanced industrial-strength implementation requires considering all three aspects as a single whole providing a physical layer that efficiently supports all these aspects. The layer is primarily based on data organization and memory management techniques. Query processing requires support for vast amounts of data that can exceed main memory and thus requires processing in secondary storage (i.e. on disk). Update processing requires a compromise between data organizations optimized for querying and updating. Using XQuery as a programming language requires fast processing in main memory without essential overheads resulting from support for external memory. In this paper we give an overview of our native XML DBMS implementation named Sedna focusing on its physical layer that provides efficient support for all the three aspects mentioned above.

The paper is organized as follows. Section 2 presents an overview of the system including its architecture, query optimization and concurrency control techniques. Section 3 describes principal mechanisms underlying the Sedna storage system, namely data organization and memory managements. In Section 4, we discuss execution of XQuery queries over the Sedna storage system. We conclude in Section 5.

[1]We have also contributed to this work by developing a set of logical optimization techniques for XQuery [5, 6, 7].

Copyright MODIS ISPRAS, 2004.

2. SEDNA

2.1 Overview

Sedna is designed with having two main goals in mind. First, it should be the full-featured database system. That requires support for all traditional database services such as external memory management, query and update facilities, concurrency control, query optimization, etc. Second, it should provide a run-time environment for XML-data intensive applications. That involves tight integration of database management functionality and that of a programming language.

Developing Sedna, we decided not to adopt any existing database system. Instead of building a superstructure upon an existing database system, we have developed a native system from scratch. It took more time and effort but gave us more freedom in making design decisions and allowed avoiding undesirable run-time overheads resulting from interfacing with the data model of the underlying database system.

We take the XQuery 1.0 language [1] and its data model [2] as a basis for our implementation. In order to support updates we extend XQuery with an update language named XUpdate. Our update language is very close to [15].

Sedna is written in Scheme and C++. Static query analysis and optimization is written in Scheme. Parser, executor, memory and transaction management are written in C++. The implementation platform is Windows.

2.2 Architecture

Figure 1 depicts the architecture of Sedna. The Sedna DBMS has the following components. The *governor* serves as the "control center" of the system. All other components register at the governor. The governor knows which databases and transactions are running in the system and controls them. The *listener* creates an instance of the *con-*

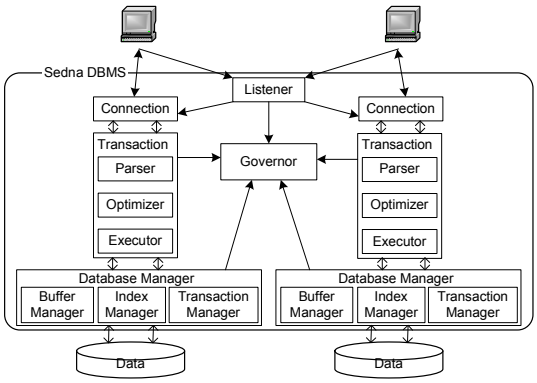

Figure 1: Sedna Architecture.

nection component for each client and sets up the direct connection between the client and the connection component. The connection component encapsulates the client's session. It creates an instance of the transaction component for each "begin transaction" client request. The *transaction* component encapsulates the following query execution components: parser, optimizer, and executor. The *parser* translates the query into its *logical representation*, which is a tree of operations close to the XQuery core. The *optimizer* takes the query logical representation and produces the optimized *query execution plan* which is a tree of low-level operations over physical data structures. The execution plan is interpreted by the *executor*. Each instance of the *database manager* encapsulates a single database and consists of database management services such as the *index manager* that keeps track of indices built on the database, the *buffer manager* that is responsible for the interaction between disk and main memory, and the *transaction manager* that provides concurrency control facilities.

2.3 Optimization

In Sedna, we have implemented a wide set of rule-based query optimization techniques for XQuery. The cost-based optimization is the subject of future work.

Function inlining technique [8] allows replacing calls to user-defined functions with their bodies. Function inlining eliminates function call overhead and allows us to optimize in static the inlined function body together with the rest of the query. This essentially facilitates the application of the other optimization techniques. We have implemented an inlining algorithm that can properly handle non-recursive functions and structurally recursive functions. The algorithm reasonably terminates infinite inlining for recursive functions of any kind that makes the algorithm applicable to any XQuery query.

Predicate push down XML element constructors [6] changes the order of operations to apply predicates before XML element constructors. It allows reducing the size of intermediate computation results to which XML element constructors are applied. This kind of transformation is of great importance because XML element constructor is an expensive operation, the evaluation of which requires deep copy of the element content. *Projection of transformation* [6] statically applies XPath expressions to element constructors. It allows avoiding redundant computation of costly XML element constructors.

Query simplification using schema is useful when a query is written by the user that has vague notion about XML document schema. Making query formulation more accurate allows avoiding redundant data scanning that is peculiar to such queries. This optimization technique is based on the XQuery static type inference.

Making query formulation as declarative as possible allows the optimizer to widen search space with optimal execution plans. The technique is the adaptation of analogous SQL-oriented technique [9] that is aimed at unnesting subexpressions into joins.

Join predicates normalization consists in rewriting joins predicates into conjunctive normal form to take the advantage of using different join algorithms, but not only nested loop join. To achieve this goal we extract subexpressions like XPath statements from join predicates and place them outside join operations, where they are evaluated only once.

Identifying iterator-free subexpressions in the body of iterator-operations reduces the computation complexity of the query by putting subexpressions which do not contain free occurrences of iterator outside the body of the iterator-operation.

2.4 Concurrency Control

Sedna supports multiple users concurrently accessing data. It uses a locking protocol to solve various synchronization problems, both at the logical level of objects like documents and nodes and at the physical level of pages.

To ensure serializability of transactions we use a well-known strict two phase locking protocol. For now, the locking granule is the whole XML document. In many cases locking of the whole XML document is not needed and this leads to the decreasing of concurrency. This is the reason why we are developing a new fine-granularity locking method, which achieves high degree of concurrency. The main idea behind our method is using numbering scheme for locking nodes and entire subtrees of XML document. The locking of entire subtree is implemented by means of locking the interval of numbering scheme labels, which includes all node labels in this subtree. This interval can be calculated by the label of the subtree root. In our approach the locking of the subtree does not lead to the locking of ancestors of the subtree root in intention mode.

3. STORAGE SYSTEM

3.1 Data Organization

Designing data organization, we would like it to be efficient for both queries and updates.

Designing data organization in Sedna we have made the following main decisions to speed up query processing. First, *direct* pointers are used to represent relationships between nodes of an XML document such as parent, child, and sibling relationships. An overview of pointers and their structures is given in Section 3.2. Second, we have developed a *descriptive schema driven storage strategy* which consists in clustering nodes of an XML document according to their positions in the descriptive schema of the document. In contrast to prescriptive schema that is known in advance and is usually specified in DTD and XML Schema, descriptive schema is dynamically generated (and increasingly maintained) from the data and presents a concise and accurate structure summary of these data. Formally speaking, every

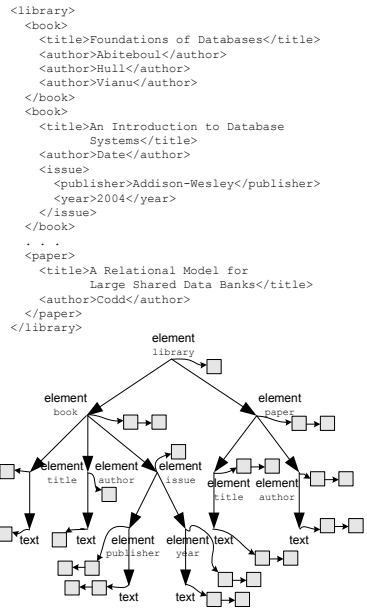

Figure 2: Data Organization.

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. As it follows from the definition, descriptive schema for XML is always a tree. Using descriptive schema instead of prescriptive one gives the following advantages: (1) descriptive schema is more accurate then prescriptive one; (2) it allows us to apply this storage strategy for XML documents which come with no prescriptive schema.

The overall principles of the data organization are illustrated in Figure 2. The central component is the descriptive schema that is presented as a tree of schema nodes. Each schema node is labeled with an XML node kind name (e.g. element, attribute, text, etc.) and has a pointer to data blocks where nodes corresponding to the schema node are stored. Some schema nodes depending on their node kinds are also labeled with names. Data blocks belonging to one schema node are linked via pointers into a bidirectional list. Node descriptors in a list of blocks are partly ordered according to document order. It means that every node descriptor in the block `i` precedes every node descriptor in the block `j` in document order, if and only if `i < j` (i.e. the block `i` precedes the block `j` in the list).

Nodes are ordered between blocks in the same list in document order. Within a block nodes are unordered[2] that reduces overheads on maintaining document order in case of updates.

In our storage we separate node's structural part and text value. The text value is the content of a text node or the value of an attribute node, etc. The essence of text value is that it is of variable length. Text values are stored in blocks according to the well-known slotted-page structure method [10] developed specifically for data of variable length. The structural part of a node reflects its relationship to other nodes (i.e. parent, children, sibling nodes) and is presented in the form of *node descriptor*.

[2]The order within a block can be reconstructed using pointers as described below.

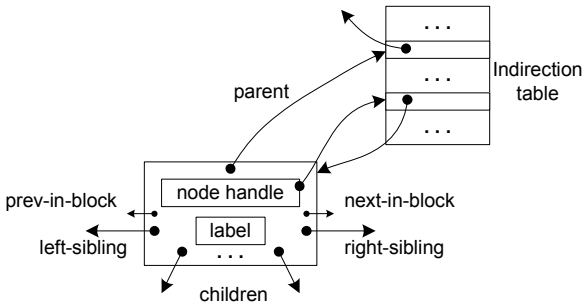

Figure 3: Common structure of node descriptor.

The common structure of node descriptors for all node kinds is shown in Figure 3. The `label` field contains a label of *numbering scheme*. Numbering scheme is discussed in Section 3.1.1. The `node handle` is discussed in Section 3.1.2. The meaning of the `left-sibling` and `right-sibling` pointers is straightforward. The `next-in-block` and `prev-in-block` pointers are used to link nodes within the block to allow reconstructing document order as was mentioned above. The `next-in-block` and `prev-in-block` pointers allow reconstructing document order among nodes corresponding to the same schema node, whereas the `left-sibling` and `right- sibling` pointers are used to support document order between sibling nodes.

Storing pointers to all children in the node descriptor may result in the node descriptor the size of which exceeds the size of block. To avoid this, we store only pointers to the first children by the descriptive schema. Let us illustrate this by the example of the `library` element in Figure 2. It has two books and one paper as child elements. In the descriptive scheme the `element library` schema node has only two children. In spite of the actual number of books and papers, the node descriptor for the library element has exactly two children pointers. These are pointers to the first `book` element and the first `paper` element. To traverse all the child `book` elements of the `library` element, we use the pointer to the first `book` element and then follow the `next-in-block` pointers (if all the children do not fit one block, we go to the next block via the interblock pointer). Having only pointers to the first children by schema allows us to save up storage space and, that is more important, to make node descriptors a fixed size. The latter is of crucial importance for efficient execution of updates because it simplifies managing free space in blocks. However, this approach leads to the following problem. If a node is inserted into the document for which there is no corresponding schema node, we have to rebuild all node descriptors in blocks belonging to the parent schema node of the inserted node. To solve this problem we maintain node descriptors to be of a fixed size only within one block by storing the number of children pointers in the header of all blocks. The number informs us that all node descriptors stored in the block have exactly this number of children pointers.

Using direct pointers is the main source of problems for efficient update execution. To make the data organization good for updates, we should minimize the number of modifications caused by the execution of an update operation. Let us consider an update operation that moves a node. This operation is invoked by the procedure of block's splitting as

a result of inserting a node into the overfilled block. If the node to be moved has children, they all must be modified to change their `parent` pointers to the node. The solution is to use indirect pointers, that is implemented via *indirection table*, to refer to the parent.

In conclusion of this section we would like to sum up the features of our data organization that are designed to improve update operations:

- Node descriptors have a fixed size within a block;
- Node descriptors are partly ordered;
- The parent pointer of node descriptor is indirect

3.1.1 Numbering Scheme

A numbering scheme assigns a unique *label* to each node of an XML document according to some scheme-specific rules. The labels encode information about the relative position of the node in the document. Thus, the main purpose of a numbering scheme is to provide mechanisms to quickly determine the structural relationship between a pair of nodes. Designing Sedna, we require the numbering scheme to provide two such mechanisms: (1) to determine the ancestor-descendent relationship; (2) to compare nodes by document order. The first mechanism allows supporting query execution plans based on *containment joins* as proposed in [11, 12]. The second mechanism is used for implementing XQuery operations based on document order (e.g. node comparison, XPath, etc.). The main problem with the existing numbering schemes for XML (for example, with that proposed in XISS [13]) is that inserting nodes may lead to the reconstruction of the entire XML document. We have developed a novel numbering scheme that does not require document reconstruction.

The idea of our numbering scheme is based on the following observation: for any two given strings `str1` and `str2` such that `str1 < str2` (compared by lexicographical order) there exists a string `str` such that `str1 < str < str2`. For example, `(str1= "abn", str2 = "ghn") => (str = "bcb")` and `(str1 = "ab", str2 = "ac") => (str = "abd")`. In our numbering scheme, the label of a node is a pair `(id, d)` where `id` is a string (called *prefix*), `d` is a character (called *delimiter*), and the string interval `(id, id+d)` (where `+` denotes string concatenation) sets the range of labels for all descendents of the given node. Labels are assigned to the nodes of a document in such a way that the following conditions are satisfied: (1) for any two given nodes `x` and `y` labeled as `(id1, d1)` and `(id2, d2)` respectively, `x` is an ancestor of `y` if and only if `id1 < id2 < id1+d1`; (2) for any two given nodes `x` and `y` labeled as `(id1, d1)` and `(id2, d2)` respectively, x precedes y in document order if and only if `id1 < id2`. For brevity, we omit the algorithm that assigns a new label for the inserted node and that does not require document reconstruction. It is worth mentioning that a numbering scheme that provide support for document order can also be used to implement the XQuery notion of *unique identity* because in such numbering scheme nodes that have equal labels are the same node. We also would like to say a few words about how we store labels that are variable-length size. If the length of a label is not larger than 8 bytes, we store it inside node descriptor (the `label` field), otherwise we store it outside the node descriptor and `label` servers as a pointer to that string. We manage 'outside' `label`s the same way as we manage string data.

3.1.2 Node Handle

Implementation of some operations and database mechanisms requires support for *node handle* that is immutable during the whole life-time of the object and can be used to access the node efficiently. For instance, hode handle can be used to refer to the node from the index structures. As discussed in Section 4, node handle is also necessary for the proper implementation of update operations. As mentioned above, execution of some update operations (e.g. insert node) might lead to block splitting that in turn results in moving nodes. Therefore, pointers to nodes do not possess the property of immutability. Although the label of numbering scheme is immutable and allows uniquely identifying the node, it requires document tree traverse to get the node by the label. Our implementation of node handle is shown in Figure 3. We exploit the indirection table that is also used to implement indirect pointers to parent. Node handle is the pointer to the record in the indirection table which contains the pointer to the node.

3.2 Memory Management

As discussed in Section 3.1, one of the key design choices concerning the data organization in Sedna is to use direct pointers to present relationships between nodes. Therefore, traversing the nodes during query execution results in intensive pointer dereferencing. Making dereferencing as fast as possible is of crucial importance to achieve high performance. Although using ordinary pointers of programming languages powered by the OS virtual memory management mechanism is perfect for performance and programming effort required, this solution is inadequate for the following two reasons. First, it imposes restrictions on the address space size caused by the standard 32-bit architecture that still prevails nowadays. Second, we cannot rely on the virtual memory mechanism provided by OS in case of processing queries over large amounts of data because we need to control the page replacement (swapping) procedure to force pages to disk according to the query execution logic [16].

To solve these problems we have implemented our own memory management mechanism that supports 64-bit address space (we refer to it as Sedna Address Space - SAS for short) and manages page replacement. It is supported by mapping it onto the process virtual address space (PVAS for short) in order to use ordinary pointers for the mapped part. The mapping is carried out as follows. SAS is divided into layers of the equal size. The size of layer has been chosen so that the layer fits PVAS. The layer consists of pages (that are those in which XML data are stored as described in Section 3.1). All pages are of the equal size so that they can be efficiently handled by the buffer manager. In the header of each page there is the number of the layer the page belongs to. The address of an object in SAS (that is 64-bit long) consists of the layer number (the first 32 bits) and the address within the layer (the second 32 bits). The address within the layer is mapped to the address in PVAS on an equality basis. So we do not use any additional structures to provide the mapping. The address range of PVAS (to which the layers of SAS are mapped) is in turn mapped onto main memory by the Sedna buffer manager using memory management facilities provided by OS.

Dereferencing a pointer to an object in SAS `(layer_num, addr)` is performed as follows. `addr` is dereferenced as an ordinary pointer to an object in PVAS. This may result in a

memory fault that means there is no page in main memory by this address of PVAS. In this case buffer manager reads the required page from disk. Otherwise the system checks whether the page that is currently in main memory belongs to the `layer_num` layer. If it is not so, the buffer manager replaces the page with the required one. It is worth mentioning that the unit of interaction with disk is not a layer but a page so that main memory may contain pages from different layers at the same time.

The main advantages of the memory management mechanism used in Sedna are as follows:

1. Emulating 64-bit virtual address space on the standard 32-bit architecture allows removing restrictions on the size of database;
2. Overheads for dereferencing are not much more than for dereferencing ordinary pointers because we map the layer to PVAS addresses on an equality basis;
3. The same pointer representation in main and secondary memory is used that allows avoiding costly pointer swizzling (i.e. the process of transformation of a pointer in secondary memory to the pointer that can be used directly in main memory is called).

4. QUERY EXECUTION

In this section we discuss XQuery query execution over the storage system described in the previous section.

The executor operates with the query execution plan that is a tree of physical operations which consume data from one another in a cascading fashion. Our set of physical operations covers the functionality of the XQuery library [3] and XQuery Core. It turns out that operating only in terms of the XQuery data model does not allow us to support the full variety of possible query execution plans. In addition to the physical counterparts of the XQuery data model notions, we use tuples. Tuple is an ordered set of atomic values or pointers to node descriptors. Using tuples allows us to extend the set of physical operations with relational-like operations such as join and group by.

The set of physical operations also provides support for updates. The statement of XUpdate is represented as an execution plan which consists of two parts. The first part selects nodes that are target for the update, and the second part perform the update of the selected nodes. The selected nodes as well as intermediate result of any query expression are represented by direct pointers. The update switches to indirect pointers presented as node handles. It is necessary because the sequential updating of the selected nodes might invalidate pointers to them by performing a number of move operations. Switching to node handles fully avoids this problem.

4.1 Query Execution Aspects

In this section we would like to emphasize the query processing aspects that are specific to our executor.

4.1.1 Element constructors

Besides the well-known heavy operations like joins, sorting and grouping, XQuery has a specific resource consuming operation - XML element constructor. The construction of an XML element requires deep copy of its content that leads to essential overheads. The overheads grow significantly when a query consists of a number of nested element constructors. Understanding the importance of the problem, we propose *suspended element constructor*. The suspended element constructor does not perform deep copy of the content of the constructed element but rather stores a pointer to it. The copy is performed on demand when some operation gets into the content of the constructed element. Using suspended element constructor is effective when the result of the constructor is handled by operations that do not analyze the content of elements. Our latest research [6] allows us to claim that for a wide class of XQuery queries there will be no deep copies at all. Most XQuery queries can be rewritten in such a way that above the element constructors in the execution plan there will be no operations that analyze the content of elements.

4.1.2 Different Strategies for XPath Queries Evaluation

Using descriptive schema as an index structure allows us to avoid tree traverse and speed up query execution. Let us consider the following XPath query: `//title`. We call it *structural path query*, because it exploits only information about structure in such a way that we do not need to make any tests depending on data to evaluate this query. Structural path queries are ideal to be evaluated using descriptive schema. We start evaluation of the query with traversing the descriptive schema for the context document (See Figure 2). The result of traverse is two schema nodes that contain pointers to the lists of blocks with the data we are looking for. Simply passing through the first list of blocks and then through the second one we may break document order, so before outputting the result the *merge operation* is performed. 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 merge operation uses labels of the numbering scheme to reconstruct document order. The computation complexity of this operation is $O(\sum_i n_i)$ comparisons of labels, where n_i is the number of node descriptors in the i-th list of blocks.

The second query `/library/book[issue/year=2004] /title` requires more effort to be evaluated. As for the previous queries we can select `/library/book` elements using the descriptive schema, then apply the predicate and the rest of the query using pointers in data. But it seems to us the following algorithm could be more attractive. Firstly, we evaluate the structural path query `/library/book/issue /year/text()`. Secondly, we apply the predicate (we select only those nodes, for which the text is equal to 2004). And at last, we apply `../../../title` to 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 potentially cuts off lots of data and then go up the XML hierarchy to obtain the final result.

4.2 Combining Lazy and Strict Semantics

All queries formulated to databases have one thing in common - they usually operate with great amounts of data even if results are small. So query processors have to (should) deal with intermediate huge data sets efficiently. To accommodate these needs the iterative query execution model, which avoids unnecessary data materialization, has been proposed. Being developed for relational DBMSs it is general enough to

be adapted for other data models. We did this for XQuery in [14]. Keeping in mind that XQuery is a functional language, iterative model can be regarded as an implementation of lazy semantics. On the other hand, it is generally accepted [17] that computation efficiency of implementation of strict semantics for a programming language is higher comparing with implementation of lazy semantics for this language. As far as we consider XQuery as a general-purpose programming language that can be used for expressing application logic, implementing lazy semantics only has bad impact on overall executor performance. To let the XQuery implementation be efficient for both query and application logic processing we combine these two evaluation models. We have developed the XQuery executor, which keeps track of amounts of data being processed and automatically switches from the lazy to strict modes and vice versa at run-time.

The query evaluation starts in the lazy mode having the execution plan constructed. The overheads of the lazy model strongly correlates with a number of function calls made during the evaluation process. The more function calls are made, the more copies of function bodies are performed. The goal is to find the tradeoff between the copying of function body and the materializing of intermediate results of functions operations. The mechanism is as follows. Every function call is a reason to switch to strict mode if the sizes of arguments are relatively small. Vice versa, the large input sequence for any physical operation in the strict mode is a subject to switch this operation to the lazy mode.

5. CONCLUSION

In this paper we have presented an overview of the Sedna XML DBMS focusing on its physical layer. Sedna is freely available at our web site [18] and readers are encouraged to download it and have a look at it themselves.

6. ADDITIONAL AUTHORS

Kostantin Antipin, Alexander Boldakov, Dmitry Lizorkin, Leonid Novak, Maria Rekouts, Peter Pleshachkov (Institute for System Programming of the Russian Academy of Sciences). Emails: antipin, boldakov, lizorkin, novak, rekouts, peter@ispras.ru

7. REFERENCES

[1] XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xquery-20031112/

[2] 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/

[3] XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/2003/WD-xpath-functions-20031112/

[4] Mary F. Fernandez, Jerome Simeon: Growing XQuery. ECOOP 2003: 405-430

[5] M. Grinev. "XQuery Optimization Based on Rewriting, 2003, Available at www.ispras.ru/~grinev

[6] M. Grinev, P. Pleshachkov. "Rewriting-based Optimization for XQuery Transformational Queries Submitted at VLDB 2004. Available at www.ispras.ru/~grinev

[7] M. Grinev, S. Kuznetsov. "Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience", In Proc. ADBIS Conference, LNCS 2435, Springer, 2002.

[8] Maxim Grinev, Dmitry Lizorkin. "XQuery Function Inlining for Optimizing XQuery Queries". Submitted to ADBIS 2004.

[9] U. Dayal. "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers", In Proc. VLDB Conference, 1987.

[10] A. Silberschatz, H. Korth, S. Sudarshan. "Database System Concepts", Third Edition, McGraw-Hill, 1997.

[11] 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

[12] S. Al-Khalifa, H. Jagadish, J. Patel, Y. Wu, N. Koudas, D. Srivastava: "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", Proceedings of ICDE 2002, San Jose, California.

[13] Q. Li, B. Moon. "Indexing and Querying XML Data for Regular Path Expressions", Pro-ceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[14] K. Antipin, A. Fomichev, M. Grinev, S. Kuznetsov, L. Novak, P. Pleshachkov, M. Rekouts, D. Shiryaev. "Efficient Virtual Data Integration Based on XML", In Proc. ADBIS Conference, LNCS 2798, Springer, 2003.

[15] P. Lehti, "Design and Implementation of a Data Manipulation Processor for an XML Query Language", Technische Universitt Darmstadt Technical Report No. KOM-D-149, http://www.ipsi.fhg.de/~lehti/diplomarbeit.pdf, August, 2001.

[16] H.-T. Chou, D. J. DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems", Proceedings of VLDB, 1985.

[17] R. Ennals, S.P. Jones, Optimistic evaluation: an adaptive evaluation strategy for non-strict programs, Proceedings of the ICFP'03, August 25-29, 2003, Uppsala, Sweden

[18] Sedna XML DBMS - http://modis.ispras.ru/Development/sedna.htm