Sedna: Native XML Database Management System (Internals Overview)

Ilya Taranov, Ivan Shcheklein, Alexander Kalinin, Leonid Novak, Sergei Kuznetsov,
Roman Pastukhov, Alexander Boldakov, Denis Turdakov, Konstantin Antipin,
Andrey Fomichev, Peter Pleshachkov, Pavel Velikhov, Nikolai Zavaritski
Institute for System Programming of the Russian Academy of Sciences
Alexander Solzhenitsyn street 25, 109004 Moscow, Russia
{taranov, shcheklein, kalinin, novak, kuzloc, ignatich, boldakov, turdakov,
antipin, fomichev, peter, velikhov, mejedi}@ispras.ru

Maxim Grinev
ETH Zurich
CAB F 78, Universitätstrasse 6
8092 Zürich, Switzerland
grinevm@inf.ethz.ch

Maria Grineva
ETH Zurich
CAB F 78, Universitätstrasse 6
8092 Zürich, Switzerland
grinevam@inf.ethz.ch

Dmitry Lizorkin
Google
Balchug street 7
115035 Moscow, Russia
lizorkin@google.com

ABSTRACT

We present a native XML database management system, Sedna, which is implemented from scratch as a full-featured database management system for storing large amounts of XML data. We believe that the key contribution of this system is an improved schema-based clustering storage strategy efficient for both XML querying and updating, and powered by a novel memory management technique. We position our approach with respect to state-of-the-art methods.

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations—*Composite structures*; D.2.11 [**Software**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Performance

Keywords

XML database, XQuery, native XML storage, numbering scheme, memory management, data multiversioning

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Flexibility of XML for representing heterogeneous data and its growing popularity make XML a basic data storage representation for universal data management systems. Early works on XML data storages were around non-native strategies that are based on mapping an XML data model onto relational or object-oriented models [24]. We believe that existing relational and object-oriented storages cannot be adequately customized to support XML flexibility to full extent. We have implemented a native XML database system, Sedna, from scratch to benefit from maximum freedom in developing proper design principles for managing XML data. The key contributions of this system are:

1. Schema-based clustering storage strategy efficient for both XML querying and updating;
2. A novel memory management technique.

This paper illustrates Sedna with the focus on storage manager features.

The paper is organized as follows. Section 2 discusses related storage strategies for XML and positions Sedna with respect to different storage aspects. Section 3 presents an overview of Sedna design and architecture. Section 4 describes the principle mechanisms underlying the Sedna storage system. In Section 5, notable aspects of XQuery execution over the storage system are considered. Sedna transaction aspects are discussed in Section 6. We conclude in Section 7.

2. RELATED WORK

Trends in native XML storage strategies can be classified into 3 aspects: (i) data organization, (ii) representation for inclusion relationship for XML elements, and (iii) memory management. Discussion on related work is consequently organized with respect to the following three aspects.

Data organization There are two main approaches to data organization. The first one is based on an

assumption that an XML element is frequently queried together with its sub-elements, so these should be clustered together [7, 15]. This approach corresponds to dividing a tree of an XML document into subtrees, and is usually referred to as a *subtree-based* storage strategy.

Another alternative is to cluster together elements of the same name that are at the same level of hierarchy in an XML tree [8]. Such clustering corresponds to organizing nodes according to a schema and is thus usually referred to as *schema-driven* storage strategy. These two approaches have mutually opposite advantages and disadvantages: subtree-based storage is efficient for retrieving an element containing subelements of different types, while schema-driven storage is efficient for retrieving only subelements of particular types instead of the whole element. Additionally, schema-driven storage is generally more computationally efficient for selecting nodes with respect to a predicate, because unnecessary nodes are not fetched from disk when the predicate is evaluated.

A combination of the two approaches is presented in [20]: XML nodes that constitute relatively integrated semantic unit are grouped into a semantic block. Nodes within a semantic block are clustered according to the subtree-based storage strategy, while semantic blocks are clustered according to a schema-driven storage approach.

In Sedna, we use the schema-driven storage strategy.

Element inclusion relationship As concerns inclusion relationship representation, one approach is to determine parent-child and ancestor-descendant containment relationships by evaluating special join operations (e.g. structural or containment joins) over labels assigned to nodes according to a numbering schema. The main advantage of the approach is that it can be implemented over a relational database that allows reusing many standard components of a database system. Another approach that is more promising with respect to query execution speed is to connect nodes via pointers. The latter has two options: (1) *location-independent* pointers, often referred to as OIDs (Object IDentity) [20, 7], or (2) *direct* pointers. Location-independent pointers remain valid even if a pointed XML node physically moves within a database (e.g. as an effect of executing updates over an XML document). On the other hand, direct pointers allow traversing an XML tree faster when executing XML queries. In Sedna, we combine location-independent and direct pointers to speed up query execution without significantly increasing update execution time.

Memory management An important aspect of supporting both location-independent and direct pointers is a cost of pointer dereferencing which depends on the nature of a pointer and its correlation with a conventional virtual memory address. A database pointer cannot be just an address in the conventional *virtual address space* (further called VAS for short). A database pointer thus has to be an address in a larger *Database Address Space* (further referred to as DAS) which in turn has to be mapped onto the conventional VAS. The cost of pointer dereferencing depends on the cost of the mapping. Reducing the cost of pointer dereferencing is undeservedly passed over in related work when XML storage strategies are considered, though it is of much importance for the overall system performance. There is a number of techniques proposed to eliminate the limitation of the VAS [25, 16]. These techniques use various schemes of *pointer swizzling* (also referred to as *pointer relocation* [16]) that implement conversion of DAS addresses to VAS addresses. The disadvantage of all of the techniques is that the pointer representations in DAS and VAS are different that makes the conversion expensive. We propose a novel technique in which both pointer representations are the same that eliminates the need for pointer swizzling.

3. SEDNA DESIGN AND ARCHITECTURE

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

Our starting decision for Sedna design was not to adopt any existing database system, since the latter would have inevitably resulted in unfavorable compromises. Instead of building an additional abstraction layer upon any existing database system, we developed Sedna as a native system from scratch. Such an approach 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 a data model of the underlying database system.

We took the XQuery 1.0 language [3] and its data model [6] as a basis for our implementation. In order to support updates, we developed our extension to XQuery with an update language named XUpdate. Our update language is syntactically close to [17].

Figure 1 depicts the Sedna architecture. Sedna has the following components. The *governor* component serves as the "control center" of the system: it keeps track of all databases and transactions running in the system and manages them. All other components in Sedna keep registered at the governor throughout all their running cycle.

For each Sedna client, the governor creates an instance of the *connection* component and establishes the direct connection between it and the client. The connection component encapsulates the client session. For each database transaction initiated by a client, the connection component creates an instance of the *transaction* component. The transaction component encapsulates components involved in query execution: parser, optimizer, and executor. The *parser* is in charge of translating a query into its *logical representation*—a tree of operations inspired by the XQuery core [5]. 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* component.

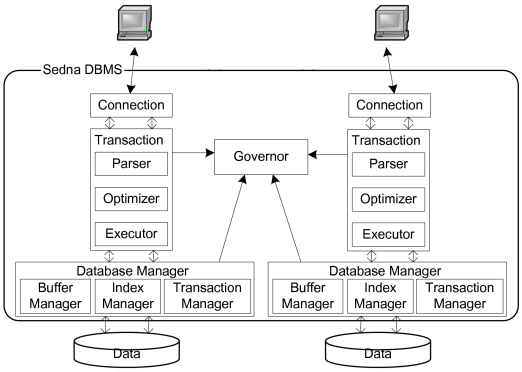

Figure 1: Sedna architecture

On the physical level, each database is encapsulated by an instance of the *database manager* that consists of the *buffer manager* that is responsible for the interaction between disk and main memory, and the *transaction manager* that provides concurrency control facilities.

4. STORAGE SYSTEM

This section discusses the two primary aspects of the storage system in Sedna: data organization and memory management.

4.1 Data Organization

Data organization in Sedna is designed with the goal of being efficient for both queries and updates.

In order to speed up query processing, the following design decisions have been made in Sedna data organization.

1. *Direct* pointers are used to represent XML node relationships such as parent, child, and sibling ones. Unlike relational-based approaches that require performing joins for traversing an XML document, traversing in Sedna is performed by simply following a direct pointer. Additional lower-level techniques are implemented to facilitate quicker pointer dereferencing, as discussed in Section 4.2.

2. We have developed a *descriptive schema-driven storage strategy* which consists of 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 or XML Schema, descriptive schema is generated from data dynamically (and is maintained incrementally) and represents concise and accurate structure summary for data. Using descriptive schema instead of prescriptive one gives the following advantages: (1) descriptive schema is more accurate than prescriptive one; (2) the storage strategy based on XML descriptive schema is applicable to any XML document, even the one that comes with no prescriptive schema.

Technically, descriptive schema employed in Sedna is a relaxed variation of DataGuides [9]: namely, every path in an XML document has exactly one path in the descriptive schema. It follows from the above definition that descriptive schema for an XML document is essentially a tree [8].

```
<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>
</library>
```

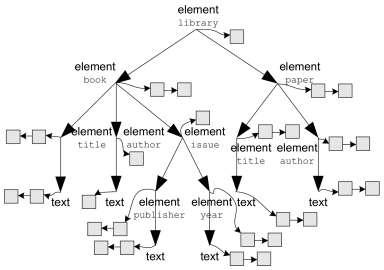

Figure 2: A sample XML document (top) and its internal representation in Sedna (bottom)

Descriptive schema serves as an entry point to the structural part of the XML document. Namely, every schema node has a pointer to data blocks that store XML nodes corresponding to the given schema node. As XQuery queries and XML update statements [14] access nodes in an XML document with XPath expressions, the descriptive schema plays a role of a naturally built index for evaluating XPath expressions.

The overall principles of the data organization are illustrated in Figure 2. The descriptive schema represented as a tree of schema nodes is the central component in the data organization. Each schema node is labeled with an XML node kind [6] (e.g. *element*, *attribute*, *text*, etc.) and has a pointer to data blocks that store XML nodes corresponding to the given schema node. Some schema nodes depending on their node kinds are also labeled with names. Data blocks related to a common schema node are linked via pointers into a bidirectional list. Node descriptors in a list of blocks are partly ordered according to document order [3]. It means that every node descriptor in the i-th block precedes every node descriptor in the j-th block in document order, if and only if $i < j$ (i.e. the i-th block precedes the j-th block in the list).

Within a block, nodes are unordered[1] for reducing the overhead of maintaining document order in case of updates. All inter-node pointers are made direct, except for parent pointers implemented via indirection table to guarantee the fixed number of pointer modifications involved into any node updating.

In our storage we separate the structural part of an XML node (i.e. markup) and text value. The text value represents

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

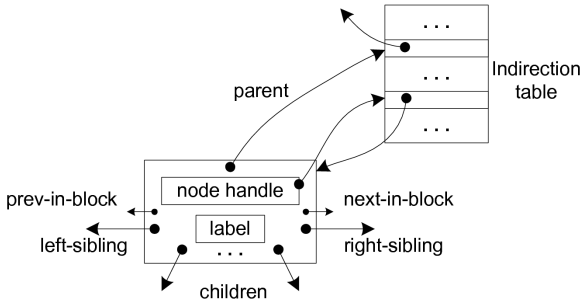

Figure 3: Common structure of node descriptor

string properties of XML nodes, in particular, the content of a text node [6] or the string-value of an attribute node. By nature of XML, text values essentially have unrestricted length, varied for different nodes in the XML document. Due to unrestricted length support required for text values, they are stored in blocks according to the well-known slotted-page structure method [23] 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*.

The structure of a node descriptor typical to all node kinds is shown in Figure 3. A node descriptor contains the following components:

- The `label` field holds a label of *numbering scheme* that is used for tracing ancestor-descendant relationships of XML nodes; details on the numbering scheme employed in Sedna are discussed in Section 4.1.1.
- The `node handle` field provides the identifier that holds the immutability property during the whole lifetime of the corresponding XML node; node handle is discussed in detail in Section 4.1.2.
- `left-sibling` and `right-sibling` are pointers to the left the right siblings of the given XML node respectively; these pointers are used for supporting document order between siblings that correspond to potentially different schema nodes.
- `next-in-block` and `prev-in-block` are pointers that link nodes within a single block to allow reconstructing document order among nodes corresponding to the same schema node.

Since the size of a node descriptor is crucial for the overall size of a database, a descriptor for an XML node contains pointers not to all its child nodes, but only to the first ones by descriptive schema. Let us illustrate this principle by the `library` element in Figure 2. For the sample XML document in that figure, the `library` element has two `book`s and one `paper` as child elements. On the other hand, the schema node for `element library` has only 2 children. Independently of the actual number of child `book`s and `paper`s, the node descriptor for the `library` element has exactly two child pointers: one to the first `book` element and one to the first `paper` element. To traverse all the child `book` elements of the `library` element, the database management system uses the pointer to the first `book` element and then follows the `next-in-block` pointers; interblock pointer are used if child nodes take multiple blocks. Additionally to saving up storage space, keeping only pointers to the first children by schema allows making all node descriptors of a given schema node be of fixed size. The latter is of crucial importance for execution of updates, because fixed size facilitates more efficient management of free space in blocks.

Fixed size of a node descriptor for each schema node requires specific handling for update operations that modify descriptive schema of an XML document. For instance, if an XML node is inserted that does not yet have the corresponding schema node, then node descriptors associated with the parent schema node should be expanded by one child pointer. Since massive modification for all these node descriptors is unacceptable, this modification is implemented in the delayed per-block fashion. Precisely, the size of node descriptors is kept the same within a single block, but is relaxed to be potentially different across different blocks associated with the same schema node. The number of child pointers is stored in the header of each block to convey information about the size of node descriptors stored in the block.

To make the data organization well suited for XML updates, our requirement was that each update over an XML node involves modifying a constant number of fields in the database. For illustration, let us consider an update operation that moves a node. This operation is routinely invoked by the procedure of splitting a block as the result of inserting an XML node into an overfilled block. For sustaining the physical parent-child consistency, the parent property [6] has to be modified accordingly for each child of the node to be moved. If a parent property was implemented as a direct database pointer, then the latter modification would have required the number of external operations proportional to the number of child nodes, which is computationally unacceptable. To avoid massive modification in this case, a parent property is implemented as an indirect pointer via the special *indirection table*. With such design, modifying the parent property for all the child nodes requires updating only a single pointer—the one residing in the corresponding entry of the indirection table.

To summarize, the following features of the data organization in Sedna are designed to facilitate execution of XML update operations:

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

4.1.1 Numbering Scheme

A numbering scheme assigns a unique *label* to each node in an XML document in accordance with some scheme-specific rules. Labels encode information about relative positions of nodes in an XML document. The main purpose of a numbering scheme is to provide mechanisms to quickly obtain structural relationship between a pair of XML nodes. In Sedna design, a numbering scheme was required to provide two such mechanisms: (1) checking the ancestor-descendant relationship; (2) comparing XML nodes by document order. The first mechanism allows supporting query execution plans based on *containment joins* as proposed in [15, 1]. The second mechanism is used

for implementing XQuery operations based on document order, e.g. node comparison, XPath expression, etc.

The main drawback of the previously existing numbering schemes for XML (e.g., the one proposed in XISS [18]) is that inserting nodes into an XML document periodically requires reconstruction of labels for the entire XML document. We have developed a novel numbering scheme that does not require such reconstruction.

The idea of our numbering scheme is based on the following observation: for any two strings S_1 and S_2 such that $S_1 < S_2$ lexicographically, there exists a third string S which is lexicographically in between the first two, i.e. the inequality $S_1 < S < S_2$ holds. For example, for S_1 = "abn" and S_2 = "ghn", S can be chosen as S = "bcb"; for S_1 = "ab" and S_2 = "ac", S can be chosen as S = "abd".

In our numbering scheme, the label of an XML node is a pair (id, d) where 'id' is a string called a *prefix*, and d is a character called a *delimiter*. Denoting string concatenation by the plus sign, the string interval $(\mathrm{id}..\mathrm{id} + d)$ specifies the range of labels for all descendants of the given XML node. Labels are assigned to nodes of an XML document to satisfy the following conditions:

1. For any two XML nodes x and y labeled as (id_1, d_1) and (id_2, d_2) respectively, the node x is an ancestor of the node y if and only if $\mathrm{id}_1 < \mathrm{id}_2 < \mathrm{id}_1 + d_1$.
2. In the same notation, the node x precedes the node y in document order if and only if $\mathrm{id}_1 < \mathrm{id}_2$.

Note that a numbering scheme that provides support for document order can also be used to implement the XQuery notion of *unique identity* [3], because two nodes have equal labels in such a numbering scheme if and only if they are the same node.

4.1.2 Node Handle

Each XML node stored in Sedna is supplied with a *node handle* that conforms to 3 requirements: (i) the node handle uniquely identifies the XML node in the database; (ii) a node handle provides quick access to the XML node; (iii) a node handle is immutable for the whole lifetime of its corresponding XML node, even if the latter is physically moved within the database.

Node handles are required for effective implementation of some database mechanisms and query and update operations. For instance, hode handle is used to refer to an XML node from index structures.

Node handlers cannot be implemented as straightforward database pointers, since a physical address of an XML node can change during the lifetime of the node. For example, an XML node can change its physical location if its containing block is split or merged with another block as a side effect of update operations. Labels of the numbering scheme, although immutable and identify XML nodes uniquely, are not suitable as node handles as well, since retrieving an XML node by its label requires an additional index upon labels.

With the above stated requirements, the node handle in Sedna is implemented as an entry of the indirection table that holds a pointer to that node. Actually indirection table lays in the same blocks the nodes lay. While a node can change its physical location, entries of the indirection table are guaranteed to preserve their position during the lifetime of the XML nodes they point to. A conceptual illustration of a node handle is given in Figure 3.

Garbage collection of node handles is done on transaction commit or database recovery, when orphaned blocks are deleted.

4.2 Memory Management

As discussed in the previous subsection, representing relationships between XML nodes with direct pointers is one of the key design decisions in Sedna data organization. Therefore, traversing between XML nodes during query execution results in intensive pointer dereferencing. For achieving high performance of the XML database management system, it is thus of crucial importance to make pointer dereferencing operation as quick one as possible. Although conventional pointers provided by programming languages and powered by a virtual memory management mechanism of an operating system are generally effective for performance reasons and the programming effort they require, this solution is inadequate for an XML database management system for the following reason: a database management system cannot reasonably rely on the virtual memory mechanism provided by an operating system for executing queries over large amounts of data, because it is beneficial to use query execution logic to control the page replacement procedure (swapping) when forcing pages to disk [4].

To handle the situation, we have implemented our own memory management mechanism that supports 64-bit address space and manages page replacement. We further refer to the proposed address space as *Sedna Address Space*, SAS for short. SAS is supported by mapping the address space onto the virtual address space of a process (PVAS) in order to use ordinary pointers for the mapped part. The mapping is carried out as follows. SAS is divided into layers of equal size. The size of layer has been chosen so that the layer fits into PVAS. The layer consists of pages (that are those in which XML data are stored as described in Section 4.1). All pages are of equal size so that they can be efficiently handled by the buffer manager. The header of each page contains the layer number the page belongs to. The 64-bit address of an object in SAS consists of the layer number (the first 32 bits) and the address within the layer (the remaining 32 bits). An address within a layer is mapped to the address in PVAS on equality basis. In such design no additional structures are required for providing the mapping. The address range of PVAS (to which layers of SAS are mapped) is in turn mapped onto main memory by the Sedna buffer manager using memory management facilities provided by an operating system.

The architecture of Sedna storage manager is shown in Figure 4. We implemented our own 64-bit virtual address space and memory management mechanism to support DAS. The key idea of memory management in Sedna is integrating persistence with virtual memory system. For achieving this integration, we suggest dividing DAS into layers of equal size that fits VAS. A layer consists of pages; pages store XML data and also have equal sizes for uniform handling by the Sedna buffer manager. The address of an object in DAS consists of (1) the layer number and (2) the address within the layer.

As shown in Figure 4, an address within the layer is mapped to the address in VAS on the equality basis: the address of an object in the VAS is the address of the object within a layer. The address range of VAS is

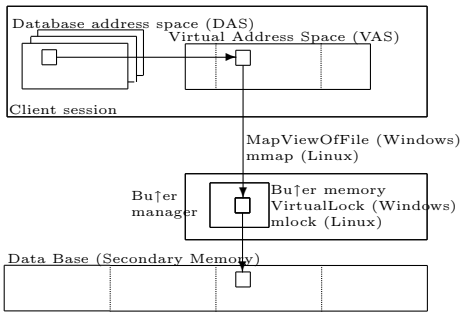

Figure 4: Sedna storage manager organization.

in turn mapped onto main memory by the Sedna buffer manager using memory management facilities provided by the operation system. The memory management strategy is implemented uniformly for both Linux and Windows; the corresponding system calls involved in buffer management are shown in Figure 4. The address mapping suggested allows dereferencing the pointer more effectively, as it eliminates pointer swizzling overhead.

Dereferencing a pointer to an object in SAS `(layer_num, addr)` is performed as follows. Address `addr` within a layer is dereferenced to an object in PVAS as an ordinary pointer. If there is no page in main memory by this address of PVAS, then dereferencing results in a memory fault. In this case the 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 addressed by `layer_num`. If it is not the case, then the buffer manager replaces the page with the required one. It should be noted that the unit of interaction with disk is not a layer but a page, so main memory generally contains pages from multiple layers at a time.

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

- There is virtually no restriction on the database size, as we provide a 64-bit virtual address space, that can be emulated on a standard 32-bit architecture.
- Overhead for dereferencing a database pointer is comparable to the one for conventional pointers, since a database layer is mapped to PVAS addresses on equality basis.
- Costly pointer swizzling is avoided by using the same pointer representation in main and secondary memory.

5. QUERY EXECUTION

Query execution in Sedna is designed with two main goals in mind:

1. Support for a wide range of queries/statements: XQuery queries, XML update statements, data definition language statements.
2. High performance for both query evaluation and updates execution.

For maximizing the benefits of Sedna internal data representation design, query optimization strategies are worked out in correspondence with this representation. For instance, since data organization in Sedna essentially provides a naturally built index over structural component of XML documents, it is only reasonable to utilize such an index for optimizing query execution.

Query processing in Sedna is implemented as a sequence of steps performed by the following components: (1) query parser; (2) static analyser; (3) optimizing rewriter; and (4) executor. Each of the components is discussed below.

The *query parser* takes a query or a statement submitted by the user and produces the operation tree. The parser in Sedna supports the following three types of queries and statements: (i) XQuery queries; (ii) XML update statements; and (iii) Data Definition Language statements (e.g. the 'create document' statement). The operation tree produced by the parser is designed to provide uniform representation for all the 3 query/statement types, that allows to use uniform techniques on subsequent steps for all the types.

The *static analyzer* accepts the parsed query and is in charge of performing the complete static analysis phase [3] with conformance to the Formal semantics [5]. At this stage, the static context of the query is initialized with XQuery Functions and Operators and augmented with query prolog declarations; the operation tree is expanded with imported XQuery modules, and all namespace prefixes, function names and variable names are resolved. If a query contains any static errors [3], these are detected and reported at this stage.

The *optimizing rewriter* and the *executor* deserve a more detailed discussion and thus considered in the following two subsections accordingly.

5.1 Optimizing rewriter

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.

In this section the optimization techniques which are implemented in Sedna are described. We researched several other techniques for optimizing query execution through rewriting, in particular, inlining for user-defined XQuery functions [11] and predicate push down XML element constructors [12].

5.1.1 Removing unnecessary ordering operations

The requirement for producing sequences of nodes in document order with duplicate nodes removed is specified by the XQuery semantics for many XQuery operations. Such a *Distinct-Document Order* semantics is expressed in Sedna query operation tree via explicit DDO operations. DDO operations decrease query execution performance, because they require the whole argument sequence to be evaluated before a any result item could be produced and thus break execution pipeline.

The idea for optimizing query execution with this respect is to remove unnecessary ordering operations. A decision on whether a given DDO operation in a query tree is redundant is made by analyzing the query tree. Namely, for each operation in the query operation tree, the following properties for the resulting sequence are recursively found out:

1. whether this sequence would already be in DDO;

2. whether it would consist of no more than a single item;

3. whether it would consist of nodes on a common level of an XML tree.

Using these properties, a DDO operation is removed if (i) either its argument is known to be in DDO, or (ii) DDO is not required for the resulting sequence. We researched these ideas in [19]; similar ideas can be found in [13], although different methods are used there.

5.1.2 Handling an abbreviated descendant-or-self path step

The abbreviated path step "`//`" that is expanded into `descendant-or-self::node()` is frequently used in practical XQuery queries. However, straightforward evaluation of this location step is extremely expensive. First, this step has bad selectivity, since it generally selects almost all nodes in an XML document. Second, it does not allow to use benefits of the descriptive schema-driven storage strategy employed in Sedna.

The idea for optimized evaluation of the abbreviated descendant-or-self step is to combine it with the next step in a location path: for example, expression `//para` is transformed into `/descendant::para`. The rewritten expression provides better intermediate selectivity and benefits of the Sedna schema-driven storage.

However, it is not always semantically permissible to combine the abbreviated descendant-or-self step with a next step; a well-known counter-example from [3] shows that "The path expression `//para[1]` does not mean the same as the path expression `/descendant::para[1]`". For the above reason, predicate expressions of the next step of the abbreviated descendant-or-self path step are analyzed. If the predicate expressions do not depend on context position and size (neither explicitly, nor implicitly), then the descendant-or-self step is combined with its next step while preserving the semantics of the original query.

5.1.3 Analyzing nested for-clauses

FLWOR-expressions in XQuery generally contain multiple iteration variables in for-clauses. Binding sequences with nested loop semantics An expression associated with an inner iteration variable is analyzed, and the associated expressions that do not depend on outer iteration variables are marked as *lazy*. Lazy associated expressions are evaluated just once, with the query semantics preserved.

5.1.4 Extracting structural location path fragments

We call a location path a structural one if it starts from a document node and contains only descending axes and no predicates. Many practical location paths contain structural fragments. These are automatically mapped to Sedna access operations over descriptive schema and can thus be executed very quickly, since they are executed in main memory.

5.2 Executor

In this section we provide an overview of XQuery execution over the storage system described above. We consider general executor design and then describe a number of optimization techniques which are specific to XQuery and to Sedna storage system.

Like in most database systems, query execution pipe in Sedna consists of a sequence of steps, among which are parser, query rewriter, physical optimizer and execution engine itself. The output of the physical optimization step is a query execution plan that is a tree of the physical operations. Each one is implemented as iterator and provides extended version of the well known "open-next-close" interface [10]. Design based on the standard interface allows easy addition of new physical operations.

Calling 'open' for the top-most operation results in its initialization (e.g. resources allocation, variable bindings) and in 'open' calls for all children. In this way the whole tree-structured query plan is initiated. Query is evaluated in a demand-driven fashion avoiding unnecessary data materialization. The method 'next' is called repeatedly for the top-most operation until the end-of-sequence indicator has been received. Finally, the 'close' call recursively releases resources.

At the present time, Sedna provides library of the physical operations which covers XQuery expressions and provides support for updates and data definition language. The statement of XUpdate-based language [17] 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 updates the selected nodes. The selected nodes as well as intermediate result of any query expression are represented by direct pointers. Since direct node pointers are essentially invalidated after a number of move operations are performed, the updated nodes are referred to by node handles.

In the following subsections, we emphasize several aspects of query processing that are specific to our executor: treatment for XML element constructors and evaluation of XPath location paths.

5.2.1 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 making a deep copy of its content that leads to essential computational and storage overhead. The overhead grow significantly for a query consisting of a number of nested element constructors.

Realizing the importance of the problem, we proposed constructor optimization strategies [22].

The first optimization strategy is called *embedded element constructors*. In case of two element constructors nested into one another within a query, the nested one sets the parent property of the constructed node to the element, created by the constructor it is nested to (when it is possible).

Another optimization proposal is *virtual element constructor*. It also does not perform deep copy of the content of constructed node, but rather stores a pointer to it. It is possible when the result of constructor is handled by operations, that do not traverse the constructed element's subtree, and do not depend on node identity or document order properties of constructed nodes.

Our recent research [12] 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.

6. TRANSACTION ASPECTS

Each statement is executed within a transaction.

Transaction can contain several statements and provides ACID support during execution. In this section we discuss Sedna transactions issues.

6.1 Using versions for data

One of the widely used techniques for concurrency control management is multiversioning. When using multiversioning, each data element may have several versions. Sedna uses snapshot-based scheme with data elements being pages [26]. Snapshot is a set of versions (one version per page) that is transaction-consistent. Logically snapshot is just a pair: (timestamp, list of active transactions). Thus, from the technical point of view, snapshot maintenance does not create much overhead. To create a new snapshot, we simply store the current timestamp and the list of currently active transactions. Snapshot-based versioning allows us not to worry about garbage collecting. Old versions are purged when they are not needed anymore, i.e. when they do not belong to any of the snapshots. This condition is checked when a new version of a page is created, so the procedure is not very time-consuming.

When transaction updates some page, a new version of this page is created. Locking scheme prevents two concurrent transactions from creating uncommitted versions of the same page. When transaction commits, all its versions become last committed ones. If it is rolled back, all its versions are simply discarded. When reading, transaction fetches last committed versions (or reads its own versions if it has created them). Versioning mechanism is transparent from user and transactional point of view, since it is encapsulated in the storage manager.

6.2 Locking scheme

Sedna uses the classical strict two-phase locking approach (S2PL) [2] to support the isolation property of transactions. This approach allows multiple users access data without paying attention to concurrency mechanism details. At the present moment, locking granularity is an XML document. In many cases, locking the whole XML document is excessive and leads to a decrease in concurrency. For this reason, we are working on a finer-granularity locking scheme.

6.3 Read-only transactions

Multiversioning allows using read-only transactions (also called queries). These transactions cannot contain update statements, but they can be executed much faster due to multiversioning. Each query reads one of the snapshots, so it obtains a consistent but possibly a slightly obsolete state of the database. At the same time, reading a snapshot allows non-blocking processing (i.e. non-S2PL) for read-only transactions. Snapshots are periodically advanced to include recent updates made to the database. Since a snapshot is a simple structure, advancement takes small amount of time.

6.4 Recovery

Durability property of transactions is guaranteed by logging and recovery mechanisms [21]. These mechanisms allow us to recover database from any kind of soft crash. All the main operations (insert node, create index, etc.) are logged using the WAL protocol. Additionally, a checkpoint may be created at some moment during execution to fixate transaction-consistent state of a database. We call such a state a *persistent snapshot*. If a database is crashed at some moment in time, two-step recovery process is initiated to restore all transactions that had been committed by the moment of the crash. During the first step, transaction-consistent state of the database is restored by converting versions belonging to the persistent snapshot into last committed ones. Then, at the second step, log is processed to redo the necessary operations of committed transactions.

6.5 Hot-backup

Sedna allows creating hot-backup copies of a database. Such backup can be made even while the database is working and performing user requests. Incremental hot-backups are also supported to provide more efficient backup of rarely updated databases. From technical point of view, all necessary files are copied at a specified destination during hot-backup. First, data file is copied. To solve the infamous "split-block" problem, additional logging is used. Second, log is fixated and its files are copied. After that, any additional files (e.g. configuration ones) are backed up. During incremental hot-backup, only log files and configuration files are copied, so incremental hot-backup reduces backup time if the number of updates is relatively small. Using incremental hot-backups, it is also possible to perform some analogue of "point-in-time" recovery by applying only the required incremental parts of the required backup.

7. 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[2] and the readers are encouraged to try it out.

8. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, 2002.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C recommendation, World Wide Web Consortium, January 2007.

[4] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 127–141. VLDB Endowment, 1985.

[5] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, and J. Siméon. XQuery 1.0 and XPath 2.0 formal semantics. W3C recommendation, World Wide Web Consortium, January 2007.

[6] M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, World Wide Web Consortium, January 2007.

[2]http://modis.ispras.ru/sedna/

[7] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native xml base management system. *The VLDB Journal*, 11(4):292–314, 2002.

[8] A. Fomichev, M. Grinev, and S. Kuznetsov. Descriptive schema driven xml storage. Technical report, Institute for System Programming of the Russian Academy of Sciences, 2004.

[9] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[10] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[11] M. Grinev and D. Lizorkin. Xquery function inlining for optimizing xquery queries. In *ADBIS (Local Proceedings)*, 2004.

[12] M. Grinev and P. Pleshachkov. Rewriting-based optimization for xquery transformational queries. In *IDEAS '05: Proceedings of the 9th International Database Engineering & Application Symposium*, pages 163–174, Washington, DC, USA, 2005. IEEE Computer Society.

[13] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in xpath. In *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2003.

[14] Institute for System Programming RAS. *Sedna Programmer's Guide*, 2003–2008.

[15] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[16] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, 1991.

[17] P. Lehti. *Design and Implementation of a Data Manipulation Processor for an XML Query Language*. PhD thesis, Technische Universität Darmstadt, August 2001.

[18] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[19] D. Lizorkin. PhD thesis, Functional Techniques for Processing XML Data, Russia, 2005.

[20] X. Meng, Y. Wang, D. Luo, S. Lu, J. An, Y. Chen, J. Ou, and Y. Jiang. OrientX: A schema-based native XML database system.

[21] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[22] L. Novak, M. Grinev, and I. Taranov. Efficient implementation of xquery constructor expressions. In *SYRCoDIS*, 2008.

[23] A. Silberschatz, H. F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[24] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative xml storage strategies. *SIGMOD Rec.*, 31(1):5–10, 2002.

[25] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 395–406. ACM Press, 1994.

[26] K.-L. Wu, P. S. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *ICDE*, pages 577–586, 1993.