Application Programming Interface for XML DBMS: design and implementation proposal

© Maria Rekouts

Institute for System Programming of Russian Academy of Sciences
rekouts@ispras.ru
Ph.D. Advisor A.N. Tomilin

Abstract

As XML becomes ubiquitous, there are a growing number of applications that utilize it. Most of such applications working with some kinds of XML storages, in particular with Native XML Databases, need to perform navigation over XML data retrieved from the database. The Application Programming Interface with efficient navigational facilities, that XML DBMS must provide these applications with, is still under discussion among XML DBMS developers. In this paper we present design and implementation of XML DBMS Application Programming Interface that provides efficient navigational facilities.

1 Introduction

XML [4] is finding its way into applications beyond those that traditionally utilize markup languages. In particular, XML is becoming popular as a data interchange notation for database-oriented applications. On the other hand the need to process and store XML has prompted researches and developers to create Native XML DBMS (NXD). Currently, there are about 20 different native XML DBMS on the market: commercial and open source. Most of them are quite raw. And some of the main mechanisms are still under discussion. API (Application Programming Interface), that NXD should provide applications with, is one of those mechanisms. In this paper we discuss design and implementation aspects of the API for a Native XML Database named Sedna [8].

We have marked out a number of basic notions that do not depend on the data model and common to databases of different kinds. While developing the design we could not pass over relational and object-oriented databases APIs, and in particular much was borrowed from the JDBC API. But in our paper the Navigational API, that is a subinterface of XML DBMS API, is of greater interest. In this paper we focus on design of XML DBMS Navigational API and present our implementation proposal of Navigational API.

Proceedings of the Spring Young Researcher's Colloquium On Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2004

2 The Sedna API

The API must provide programmatic access to XML data from some programming language. We chose Java for this language because of the number of reasons: mostly because of its flexibility and opportune mechanisms such as garbage collection, also because of its popularity, some of the reasons we will discuss below. Using the API, application written in the Java programming language can establish a connection with DBMS server, manage user sessions, manage transactions, pass query as a string to server for execution, get the result and navigate over the result of a query.

We do not propose exact list of all API interfaces and methods but below we consider API functions in more detail. When we say object of an interface we mean an object of a class that implements that interface.

2.1 Establishing a connection

`DatabaseManager` interface is an entry point. The application starts its work with DBMS by calling static method `getConnection` of the `DatabaseManager` interface.

The API defines the `Connection` interface to represent a connection to a DBMS server. The `Connection` can also be considered as a user session. Using static method `getConnection(String DBName, String user, String password)` of the `DatabaseManager` interface, application connects to the DBMS server and the server performs authorization with the parameters specified. If authorization succeeds the object of `Connection` interface is returned. When finishing its work with DBMS application can break the connection using `close()` method of a `Connection` interface.

2.2 Managing Transaction

Ones a connection has been established, an application using the API can execute queries and updates against the DBMS server in defined transaction boundaries. Connection consists of transactions, which are held one after another. Sedna API

allows specifying transaction attributes by providing methods in the Connection interface: `begin()`, `commit()`, `rollback()` and `createStatement()`.

2.3 Managing Statements

To execute XQuery statements and to retrieve results application can use `Statement` interface. `createStatement()` of a `Connection` interface is used to get object of the `Statement` interface. `Statement` interface provides two methods for executing XQuery queries: `executeQueryLite` and `executeQueryHeavy`. `executeQueryLite(String query)` is used to get the result in a serialized form (as a String). `executeQueryHeavy(String query)` returns an object of a Sequence interface as a result. `Sequence` represents the result of XQuery query evaluation. It provides navigational facilities over XML data and its mechanisms we will consider properly.

2.4 Navigational API

Today most applications that utilize XML data stored in XML DBMS need to be provided with easy and versatile navigational facilities over that XML data. That is, XML DBMS API must include Navigational API. Such an API must allow the application to navigate over XML data that was retrieved as a result of a query evaluation.

Today Document Object Model (DOM) [1] is one of the most popular APIs for navigation over XML data. DOM is a prevailing W3Cs proposal that is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure of documents. Because of its prevalence it is also often considered as an interface for navigation over XML data retrieved from XML database. But in our API proposal we do not use DOM because of the following reasons: first, it does not support data types of the XML Schema. Second, it is designed to model XML documents, but not the result of evaluation of XQuery expression (this we discuss in 2.4.2 of this paper).

We believed, that the ways of navigation over some data (or the Navigational API functionality) are determined mostly by the data model of those data. Thus, the Sedna XML DBMS API must provide functionality for navigation over XML data that rests upon XML Data Model. So, Navigational API is the part of the whole XML DBMS API that radically differs from API for DBMS of another kind. Its design demands a new approach.

In this section we will discuss our proposal for navigational API over XML data. But before that, let us consider the issue in case of relational data in order to explore that experience for our goals.

2.4.1 Navigation over relational data using the JDBC API

The JDBC API [7] provides programmatic access to relational data from the Java programming language. Using the JDBC API, applications written in the Java can execute SQL statements and retrieve results.

To navigate over retrieved data application uses the ResultSet interface of the JDBC API. ResultSet objects can have different functionality and characteristics. These characteristics are result set type, result set concurrency and cursor holdability. The type of a Result Set object determines the level of its functionality in two main areas: (1) the way in which the cursor can be moved and (2) how concurrent changes made to the underlying data source are reflected by the ResultSet object. We consider only the first item in this section because it is relational data model specific.

The ResultSet object is most often created as the result of executing SQL statement. A ResultSet object maintains a cursor, which points to its current row of data. When a ResultSet object is first created, the cursor is positioned before the first row. The following methods can be used to move the cursor:

- `next()`— moves the cursor forward one row,
- `previous()`– moves the cursor backwards one row,
- `first()`– moves the cursor to the first row in the ResultSet object,
- `last()`– moves the cursor to the last row in the ResultSet object,
- `absolute(int row)`– positions the cursor on the row-th row of the result set object.

After the cursor is set to the row required the values of the column can be retrieved. The ResultSet interface provides methods for retrieving the values of the columns from the row where the cursor is currently positioned. Two getter methods exist for each JDBC type: one that takes the column index as its first parameter and one that takes the column name or label.

From this short JDBC review we can conclude that the navigation over relational data is completely simple because of its flat structure. First, you position a cursor at the row required; second, you retrieve a value from the column specified. In addition, we take into account the fact that in up-to-date relational DBMS the length of the row is often restricted by the length of the server physical data pages. Thus, such navigation is easy to implement as every item of the ResultSet object – the row – need not to be divided into parts and may be processed as one unit of data.

2.4.2 Navigation over XML data

In this section we consider the issue for the XML data. Here we fully rely on the XQuery 1.0 and XPath 2.0 Data Model [14].

The Navigational API of the Sedna XML DBMS must implement the notions and concepts of the XQuery 1.0 and Xpath 2.0 Data Model. In this

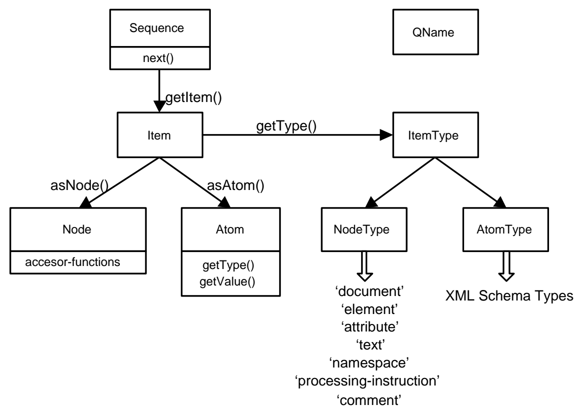

Figure 1: Interrelation between the interfaces

section we consider the data model and present the following interfaces that implement its notions and concepts: Sequence, Item, Node, Atom, ItemType, AtomType, NodeType and QName. We also consider node accessors as a mechanism for navigation. Figure 1 shows the interrelation between the interfaces.

According to [14] the value of an XQuery query expression is a sequence of zero or more items. An item is either a node or an atomic value. A node is one of seven node kinds: document, element, attribute, text, namespace, processing instruction, and comment. An atomic value encapsulates an XML Schema atomic type and a corresponding value of that type. A sequence is an ordered collection of nodes, atomic values, or any mixture of nodes and atomic values. A sequence cannot be a member of a sequence. A single item appearing on its own is modeled as a sequence containing one item.

Thus, turning back to our Sedna XML DBMS API, calling executeQueryHeavy method of a Statement interface produces the object of a Sequence interface.

The Sequence interface represents the sequence as it is defined above. Similar to the JDBC API navigation over the sequence can be of two kinds: first, you can iterate over sequence items; second, you can navigate over the Item itself.

For iterating over items of the sequence method next() of the Sequence interface is provided. Next() returns false if the sequence has finished. If true is returned the current item can be accessed using method getItem() that returns the object of the Item interface and navigation over this item can be processed.

Item interface represents the item of a sequence as it was defined above. It is a superinterface for Atom and Node interfaces. It provides methods getType() and isNode() that allows to determine whether the current item is a node or an atomic value. Method isNode() returns true if the current item is a Node and false if it is an Atom. Method getType() returns an object of an ItemType class.

ItemType interface is a superinterface for AtomType and NodeType interfaces. An object of the AtomType interface describes one of the XML Schema atomic types [13]. An object of the NodeType interface describes one of the seven node kinds: document, element, attribute, text, namespace, processing instruction, and comment.

When the type of item has been determined a node or an atomic value can be retrieved using asNode() and asAtom() methods. asNode() returns an object of the Node interface, asAtom() returns an object of the Atom iterface.

Atom interface represents an atomic value. The object of Atom class encapsulates the AtomType field that describes the type of the atomic value, and the value.

In order for applications to be able to operate on instances of the data model, the model must expose properties of the items it contains. The XQuery 1.0 and XPath 2.0 Data Model does this by defining a family of accessor functions. We consider accessors as a mechanism for navigation over XML data and associate a corresponding method in the Navigational API with every accessor. In XQuery 1.0 and XPath 2.0 Data Model a set of accessors is defined on all seven kinds of Nodes.

Thus, in the Sedna Navigational API we define a Node interface to represent the node as it is defined in [14]. The Node class encapsulates its type and provides methods that implement accessors according to this Node type.

Below we give an overview of the methods of Node class that implements accessors with some explanations. When we use the term QName we mean the QName class of the API that encapsulates the pair of values consisting of a namespace URI and a local name. Listing all the accessors with their behavior for every kind of nodes is out of this paper size. Our proposal fully relies on [14] and all the details can be found there.

- `base-uri` returns an object of the Sequence class containing zero or one reference. Document, element, and processing-instruction nodes have a base-uri property. The base-uri of all other node types is the empty sequence (an object of the Sequence class containing zero items). If the base-uri property of a document, element, or processing-instruction node is nonempty, its value is returned. If the accessor is called on a node that does not have a base-uri property, or whose base-uri property is empty, the base-uri of that node's parent is returned. If the node has no parent, the empty sequence is returned.

- `node-kind` - returns a string value identifying the kind of node on which the accessor was called. One of the following values is returned:
  - "document" for document nodes.
  - "element" for element nodes.
  - "attribute" for attribute nodes.
  - "text" for text nodes.
  - "namespace" for namespace nodes.
  - "processing-instruction" for processing instruction nodes.

- "comment" for comment nodes.

- `node-name` returns a sequence of zero or one QNames.
  - For element and attribute nodes node-name returns the qualified name of the element or attribute.
  - For processing-instructions nodes, node-name returns a QName with the processing instruction target name in the local-name and no namespace URI.
  - For namespace nodes, node-name returns an empty sequence.
- `parent` returns a sequence containing zero or one nodes. For nodes that have a parent, parent returns the parent node. For all other nodes, it returns the empty sequence. If the return value is not the empty sequence, it will always be either an element node or a document node.
- `string-value -` every node has a string value; the way in which the string value of a node is computed is different for each kind of node and is specified in the [14]. The string value of an atomic value is computed by casting it to a string.
- `typed-value` for element nodes and attribute nodes returns the typed value of a node, which is a sequence of zero or more atomic values derived from the string value of the node and nodes type in a way that is specified in [14].
- `type -` returns the name of the type of the node (as a string) if it has one. If the type is anonymous, or if no type information exists, the name returned is no-type. For text nodes type returns untypedAtomic. For other nodes kinds, it returns an empty string.
- `children -` returns a sequence containing zero or more nodes. For document and element nodes, it returns the nodes that are the children of that node in document order. It returns the empty sequence for document and element nodes that have no children. If children exist, they will always consist exclusively of element, processing-instruction, comment, and text nodes. Attribute, namespace, and document nodes can never appear as children. For all other nodes, it always returns the empty sequence.
- `attributes -` returns a sequence containing zero or more attribute nodes. For element nodes, these are the attributes of the node. For all other nodes, it always returns the empty sequence.
- `namespaces -` returns a sequence containing zero or more namespace nodes. For element nodes, these are the namespaces of the node. For all other nodes, it always returns the empty sequence.

In this section we have presented our proposal of Navigational API for XML data. Because of the informality of our presentation some points could remain unclear, and so in the next section we give an illustrative example of the application program that uses Sedna XML DBMS API. The example will also be a material for discussion of the implementation of the API that is the second part of our paper.

3 Example Program

In this section we provide the example of application program that uses Sedna API to access the DBMS. Suppose there are documents named persons and pets in the database.

persons:

```
<persons>
   <person>
    <name>John</name>
    <nick>Joe</nick>
   </person>
   <person>
    <name>Mike</name>
   </person>
   <person>
    <name nick="Tom">Tomas</name>
   </person>
   <person>
    <name>Mark</name>
    <nick>Mark</nick>
   </person>
   <person>
    <name nick="Sam">Samuel</name>
   </person>
</persons>
```

A person can contain an information about its nick: 1) as an attribute nick of element name 2) as an element nick.

pets:

```
<pets>
  <cat>Tom</cat>
  <dog>Sam</dog>
</pets>
```

The example application has to retrieve all the persons and pets data, determine the nick of every person if there are any and the nick of every pet and prints out the equal nicks. That is, as a result of its work application prints out the persons nicks that are pet names.

```
import ru.ispras.sedna.driver.*;

String comp = computerName;
String dbName = xmark;
String login = user;
String password = pasw;

Connection con = DatabaseManager.
 getConnection(comp, xmark, user, pasw);
con.begin();
Statement st1 = con.createStatement();
```

```
Statement st2 = con.createStatement();

Vector person_nicks = new Vector();

Sequence seq1 =
 st1.executeQuery(document(\person\)/*/person);
Sequence seq2 =
 st2.executeQuery(document(\pet\)/*/*/text());

while(seq1.next())
{
 Item item = seq1.getItem();
 if (item.isNode())
 {
 Node node = item.asNode();
 Sequence children = node.children();
 while(children.next())
 {
  Item i = children.getItem();
  if (i.isNode())
  {
   Node child = i.asNode();
   if (child.node-name().equals(nick))
   {
    String nick = child.string-value();
    Person_nicks.addElement(nick);
   }
   if (child.node-name().equals(name))
   {
    Sequence attrs = child.attributes();
    while(attrs.next())
    {
     Node at = ((Item)attrs.getItem()).asNode();
     If(at.node-name().equals(nick))
     {
      String nick = at.string-value();
      Person_nicks.addElement(nick);
     }
    }
   }
  }
 }
 }
}
while(seq2.next())
{
  Atom atom = ((Item)seq2.getItem()).asAtom();
  String pet_nick = atom.value();
  if (person_nicks.contain(pet_nick))
       System.out.println(pet_nick);
}
```

4 Implementation

In this section we present our proposal for implementation of the XML DBMS API described in this paper. Before going into detail we consider the basic assumptions.

4.1 Assumptions

According to the common terminology under the term *driver* we understand the implementation of the DBMS API.

Our work is based on a client-server architecture. There are an XML database server and client applications running on the same computer or on different computers in LAN. Client applications use driver to work with XML database server.

We assume that client applications are poorly resourced, particularly in memory resources. While working with DBMS client applications can obtain big portions of data for navigation. As clients are critical in their resources the situation when a portion of data is too large for clients memory is quite reasonable. Thus, our implementation must provide some mechanisms that would allow poorly resourced clients to navigate over large portions of XML data.

4.2 Caching Model

Client applications navigation over big portions of XML data that is not in memory at the client leads to network and/or server bottlenecks. Such bottlenecks can arise due to the volume of data requested by the clients. *Caching data items* in memory at the clients can reduce the volume of data that must be obtained from server. Therefore, in our implementation we use *client data caching technique* as a fundamental technique for improving the performance and scalability of database systems [6].

There are two basic types of client caching: *intra-transaction* caching refers to the caching of data within a single transaction; *inter-transaction* caching allows clients to keep data cached locally even across transaction boundaries. Intra-transaction caching requires only that a client application be able to manage its own buffer pool (cache). This is because the transaction mechanisms will ensure that any data that has been accessed by a transaction (and hence, brought into a clients memory) is valid. In contrast, inter-transaction data caching raises the need for a cache consistency protocol to ensure the application always see a consistent (serializable) view of the database. This greatly complicates the issue [5].

In this paper we consider the implementation of intra-transaction caching technique as it is easily implemented that allows us to focus on navigational aspects.

4.3 Representation of Data Items in Client's Cache

According to the intra-transaction caching technique application obtains data items from the server during a transaction, stores some of the data items in its local cache, and when the transaction commits, the cache flushes. Thus, cache always contains the data that was obtained by the current transaction only. Now we consider how the data items are represented in client cache. As it is shown in our example in Section 3, application uses the `executeQuery` method to pass the XQuery query to the server for execution. If the execution succeeds application obtains the Sequence as a result. When an object of the Sequence interface is returned, the portion of the first item (or

the whole item if it is smaller than specified portion) is shipped into the applications cache.

In cache XML data is represented as a set of objects. These objects form trees referring one to another. The kind of object references we discuss in the next section. Each object represents a node of a tree or an atomic value and can be associated with the Node or Atom the elements of the Navigational API.

The atomic value nodes are simple, they contain the value and the type of the value and do not refer to other nodes.

The objects that are nodes have more complicated structure. Each object that is a node is one of the seven node kinds: document, element, attribute, text, namespace, processing instruction, and comment. It contains all the information that is needed to implement the accessors-functions that are provided by the Node interface of the API:

- Base-uri property;
- Kind of a node;
- The value of a node;
- The type of a node;
- Reference to a parent node;
- Reference to a first child node;
- Reference to a next sibling node;
- Reference to a first attribute node;
- Reference to a first namespace node.

The values of these fields depend on the node kind and on the node itself. That is, for example, document nodes has `NULL` reference to parent node, attribute nodes has `NULL` reference to a first child node and their reference to a sibling node is a reference to the next attribute node.

Note, every node has reference only to the first child (attribute, namespace). Due to the reference to a next child (attribute, namespace) every node of a tree is reachable, i.e. these references is enough for exhaustive tree traversal. Such a node structure provides an advatages: first, each node is of a fixed structure; second, each node is of a compact structure, since it does not contain references to all children (attributes, namespaces).

By means of the references containg in each node every node of a tree is reachable directly or indirectly via the *parent*, *children*, *attributes* or *namespaces* accessors-functions.

4.4 Unique Object Identifiers

While designing this implementation we confronted with such questions as: how to determine what piece of the data from the server is in cache, what is the objects reference to an object that is not in cache? All these leads to the need to associate with an object its *unique object identirier* (OID).

OID is used to identify the object uniquely and to implement inter-object references. It is independent of the state of the object, and it is not changed during the whole life-time of the object.

Basically, there are two kinds of OIDs: physical and logical OIDs [2, 3]. A physical OID is constructed in such a way that it contains the permanent address of the object it refers to (i.e., the id of the disk, the page number and the slot). An object can directly be loaded from server on the basis of a physical OID. On the other side, the reorganization and reclustering of the database is difficult because an object cannot simply be moved to another place if physical OIDs are used. To move an object, a placeholder must be established; but, these placeholders annihilate the advantage of physical OIDs as often two page faults are required to read an object on the basis of a physical OID and the placeholder, and the storage utilization is reduced as the placeholders of moved objects fragment the data pages.

In Sedna XML DBMS logical OIDs are used: every node of XML data on the server is associated with its logical OID. Logical OIDs are more flexible than physical OIDs, since they do not contain the permanent address of the object they reference. Objects can, therefore, be moved freely, and thus, the database can well be reorganized if logical OIDs are used. There are number of techniques to implement logical OIDs [2], in this paper we will not go into dicussion of how they are implemented in Sedna DBMS. So, in our implementation of the Sedna API we rely on the fact that logical OID is provided for a node by the server.

Thus, every object in client applications cache contains its OID. The object reference to another object, that was talk about in section 4.3, is the OID of the referenced object.

If the object is in cache, application needs to locate it in the cache [9, 3]. The *cache object table* (COT), in which all objects that are in cache are registered, realizes the mapping from OIDs to client main-memory addresses.

4.5 Application Iteraction with Cache Objects

In this section we consider what caching means for application navigating over a big portion of data using API and how it iteracts with cache objects.

Application working with API can obtain an object of Node class (see example from Section 3). Object of Node class encapsulates an OID of the corresponding node (see figure 2). The object of Node class implements accessors-functions working with cached object through COT. Having OID, it reaches the node from cache through COT and obtaines all the information, contained in cached object, needed to implement accessors-functions. Thus, application iteracts with cache object indirectly, through the COT.

Such indirect iteraction has an important advantage. Application can operate with Node objects equally, and its work is independent of the fact if the object is in cache at the moment or has been

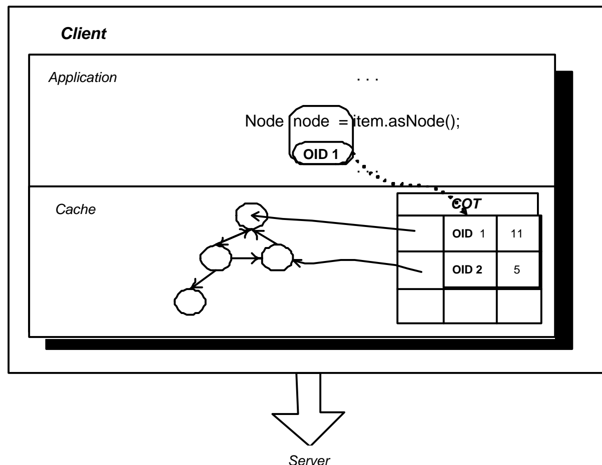

Figure 2: Application Iteraction with Cache Objects

flushed. Every time when application operates with its object of the Node class it consults the COT, and, if the COT doesnt contain the OID of the object, this object is shipped from the server into the applications cache.

To illustrate aforesaid, we return to our example.

```
Sequence seq1 =
  st1.executeQuery
   (document(\'person\')/*/person);
```

As a result of query execution application obtains the object of Sequence class. That means a portion of first item of the result sequence is shipped into the applications cache.

```
Item item = seq1.getItem();
If (item.isNode())
 {
   Node node = item.asNode();
 }
```

Here application is provided with an object node of the Node class. It encapsultes OID and iteract with cache object through COT.

4.6 Cache Flush

For the sake of completeness we consider cache flush mechanism in this section.

During application navigation over data, the data items are shipped from the server into the applications cache. At some moment cache become full and to put a new piece of data into the cache another piece of data must be flushed from the cache. Such replacement performs according to some replacement policy. In our implementation we use common cache replacement policy named LRU (*Least Recently Used*).

According to LRU policy the objects that are least recently used are flushed from the cache. For this purpose we add a column to our COT. In that column the number of object usage by application is registered. That is, every time the application iteracts with the cache object through COT, the index of this column increases by one. When the cache is full, the objects that have minimal indexes of usage in COT are flushed.

5 Related Work

From the works related to the issue discussed in the paper, we would like to mention XML:DB API [12, 10]. XML:DB Working Group has made an effort to develop API for XML DBMS that would become a common standard for XML DBMS like JDBC for relational DBMS. A few XML DBMS now provide their implementation of the XML:DB API, but in whole XML:DB API has not made a great impression on the XML database developers.

In the Sedna XML DBMS we do not use this XML:DB proposal because of the reason that it provides DOM as navigational subinterface. Why DOM does not suit for navigation over XML data we have discussed in 2.4.

6 Conclusion

In this paper design and implementation of API for XML DBMS are provided. The focus was made upon the navigation over XML data. We discussed the features of the navigation over XML data, compared them with those for relational data and on this basis provided the design of Navigational API as part of the whole API. We also provided our proposal for implementation the Navigational API for poorly resourced clients. This proposal is used in the Sedna XML DBMS.

References

[1] "Document Object Model (DOM)", World Wide Web Consortium, http://www.w3.org/DOM/.

[2] Andre Eickler, Carsten A.Gerlholf, Donald Kossmann, "A Performance Evaluation of OID Mapping Techniques". Proceedings of the 21st VLDB Conference Zurich, Switzerland, 1995.

[3] J. Eliot, B.Moss, "Working with Persistent Objects: To Swizzle or Not To Swizzle".

[4] "Extesible Markup Language (XML) 1.0", W3C Recomendation. 2nd edition (2000), http://www.w3.org/TR/2000/REC-xml-20001006

[5] Michael J. Franklin, Michael J. Carey, Miron Livny, "Transactional Client-Server Cache Consistency: Alternatives and Performance".

[6] Michael J. Franklin, Donald Kossmann, "Cache Investment Strategies".

[7] "JDBC 3.0 Specification", Sun Microsystems, Inc. October 2001.

[8] M. Grinev, A. Fomichev, S. Kuznetsov, K. Antipin, A. Boldakov, D. Lizorkin, L. Novak, M. Rekouts, P. Pleshachkov, "Sedna: A Native XML DBMS", Submitted to International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P), 2004.

[9] Alfons Kemper, Donald Kossmann, "Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis", VLDB Journal, 4, 519-567 (1995), Malcom Atkinson, Editor.

[10] Kimbro Staken, "An Introduction to the XML:DB API", http://www.xml.com.

[11] Lien Hua Chou, Herry Hamidjaja and Hongyan Yang, "Caching Techniques in Object Oriented Database".

[12] XML:DB Initiative for XML Databases, "Application Programming Interface for XML Databases", http://www.xmldb.org/xapi/index.html.

[13] "XMLSchema Part 2", World Wide Web Consortium, 2002.

[14] "XQuery 1.0 and XPath 2.0 Data Model", W3C Working Draft. 02 May 2003.