OrientX : A Schema-based Native XML Database System

Xiaofeng Meng, Yu Wang, Daofeng Luo, Shichao Lu,
Jing An, Yan Chen, Jianbo Ou, Yu Jiang
Information School
Renmin University of China, Beijing, 100872, China
xfmeng@ruc.edu.cn

ABSTRACT

The increasing number of XML repositories has provided the impetus to design and develop systems that can store and query XML data efficiently. Non-native XML method could not adequately be customized to support XML, native XML database will be more efficient. Most of existed XML database systems claimed that their systems are schema-independent. But we argue that schema plays an important role in managing XML data. In this paper, we introduce OrientX, a schema-based native XML database system. Schema information is fully investigated to enhance the efficiency and reliability in every module, including the multi-granularity native storage strategy, the cost-based XML optimizer, the schema-based path index , the XML update, and the XML security, etc.

Keywords

XML, database, native, schema

1. INTRODUCTION

XML has self-describing characteristic, supports user defined tags, and is quickly becoming the new standard for data representation and exchange in the Internet. The increasing number of XML repositories has provided the impetus to design and develop systems that can store and query XML data efficiently. Many alternatives to manage XML document collections are mapping the data into traditional database systems[2]. This introduces additional layers between the logical data and its physical storage, slowing down both updates and query processing. None of the existing DBMS could adequately be customized to support XML, despite all claims of their vendors. Another alternative approach for XML data management is designing a native XML database system. In native XML database, XML data is stored directly, retaining its natural tree structure, and queried directly, processing query statements described by XML query languages such as XQuery/XPath on the tree structure. According to the XML schema from the bottom up, native method avoids the mapping operations Some native XML databases have appeared, for example, Timber[4], Natix[5] and so on. Most of these systems are schema-independent, that is to say, the schema of the data are not required of the system. But, OrientX believes that schema plays an important role in managing XML data and is indispensable. In fact, making good use of schema information could improve the efficiency of storing and retrieving XML a lot. In this paper, we describe OrientX, an efficient, schema-based native database for storing, querying and managing XML data. The key intellectual contribution of this system is to use XML schema in managing XML. We sufficiently use schema information in all the modules, including:

- OrientX supports clustering storage strategy based on schema. The schema can also be used as a guide to choose the proper storage strategy automatically.
- Path index is built according to the schema by summering all paths in the data.
- Optimizer collects the statistical information by integrating the schema with the histograms.
- Schema information can be used to validate checking in query or updating processing.
- OrientX supports role based access control. The tree of roles has the same structure with the schema information, each role corresponding to a node in the schema.

The rest of the paper is organized as follows. We give an overview of OrientX in Section 2. In Section 3, we describe our multi-granularity native storage strategy. Section 4 focus on path index of OrientX. Section 5 and section 6 describe the query evaluator and query optimizor. Section 7 describe a novel role based access control for OrientX. We review related work in Section 8 and conclude in Section 9.

2. ARCHITECTURE OF ORIENTX

OrientX adopts client-server architecture. Client provides graphical interfaces for user managing and retrieving data. Server provides an API interface to access database. The communication between them is implemented by socket technique. The overall architecture of OrientX is shown in Figure 1. We introduce in brief some modules here, and some important modules are focused on in the following sections.

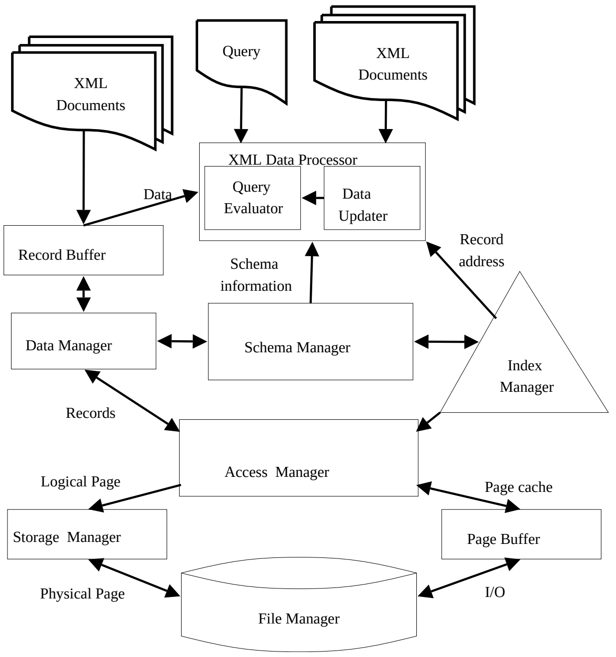


Figure 1: OrientX Architecture.

File Manager: The underlying file manager communicates with file system to create, delete, open and close data files, in units of fixed size such as 8 MB.

Storage Manager: The storage manager manages the storage space of the file in units of a physical page, which is set to 8 KB. The main tasks include: apply/free physical page, create/delete dataset, etc.

Buffer Manager: There are two layers of our Buffer Mechanism: the lower layer is page buffer, and the higher layer is record buffer. Like RDBMS, page buffer manager managing the physical pages with LRU(Least Recently Used) method. Unlike RDBMS, the record in OrientX is tree structure, and need to be generated from the byte stream, which may cost some CPU time. Record buffer cached such tree structures to reduce the generating time. Another main target of record buffer is to enable OrientX query large documents. Through record buffer, documents can be read in peaces(records), and the unoccupied record can be freed to accommodate new records. In OirentX system, the record buffer is called *tree-flog*, which means the current cursor can jump from records to records on the XML tree.

Access Manager: The access manager provides a uniform access interface to data manager, index manager, and schema manager. Details of the buffer manager and storage manager are hidden.

Data Manager: The data manager provides functions for importing, exporting, and retrieving the root of a document, etc. It formats a record(memory object) into (and from) a byte-stream.

Schema Manager: Schema-independent system can import XML data without schema. But for accelerating query processing, the system need to extract the schema form the data. That may make the schema even more huge and complex than the data. Moreover, the schema has not the function of constraining data, which will limit the use cases of schema, such as type checking in query and update. Like traditional database, OrientX is schema-based. Schema strictly constraint the type and structure of data. So, data retrieving, updating and storing are all under the schema's guidance.Schema information can be used in data layout, in choice of index, in type checking, in user access control, and in query optimization. Schema in OrientX is consistent with the XML Schema standard. Schema information is stored as a special data set in the database. Meanwhile, schema saved by tree structure is semi-structure itself, so it can restrict XML data without breaking features of XML data. Schema manager provides a uniform interface for other modules to access the schema information.

Data Processor: The data processor includes *query evaluator* and *data updater*. The former will be described in Section 5. Now we introduce the later in brief. In RDBMS, relationship between the records is represented by foreign key, and in OODBMS, relationship between objects is represented by object containment. While XML supports both of them: identity reference and nesting structure. OrientX keeps the reference integrity within updating. While deleting a complex element, all of the nested elements and values will be removed. While deleting an element referenced by other elements, the corresponding reference will be found by the value index and then deleted. The deleting of reference directly is also supported.

In our storage prototype, the elements are stored as variable length records. Each record has its parent record's or neighbor sibling record's pointer. The records may change their address because of increase or decrease contents during update operations, thus leads to the changes of the pointer. In order to decrease the modification of the pointers we introduce the oid(object id). Each element has a unique id. We use the oid table to store the oid and its corresponding storage address. In the system the record stores its parent and children oid as the pointer rather than their storage address. Therefore if the storage address of one record is changed due to update, we just to update the oid table.

To decrease the address modification of the updating record, we set a preserve factor of each page to preserve space for updating record. We supply garbage collection mechanism for space reuse.

3. ORIENTSTORE: A MULTI-GRANULARITY STORAGE STRATEGY

Several native storage strategies have been developed in [4,5,7,9,13]. According to the granularity of the records, these storage methords can be classified into Element-Based (EB), Subtree-Based (SB) and Document-Based (DB). Both the Lore system [7] and TIMBER [4] utilize the classic EB strategy, where each element is an atomic unit of storage and is organized in a depth-first traversal manner. Natix [5] is a well-known SB strategy. It divides the XML document tree into subtrees according to the physical page size, such that each subtree is a record. The sizes of the subtrees are kept as close as possible to the size of the physical page. A split matrix is defined to ensure that correlated element nodes remain clustered. Similar to the EB strategy, the records are stored in a depth-first traversal way. The storage module in the Apache Xindice system [13] employs the DB strategy, whereby the entire XML document constitutes a single record. All the above native storage strategies are schema-independent, that is, they do not leverage the power of

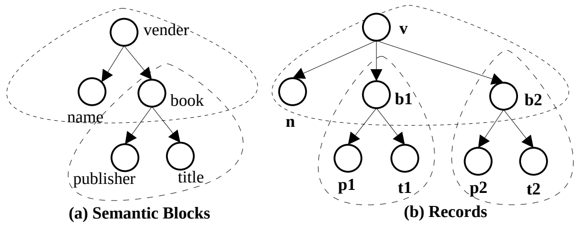


Figure 2: CSB Storage.

Schema information(XML Schema or DTD) even when such information is available. The availability of schema information is crucial to data exchange applications, and query optimizations. We observe that schema information also plays a key role in designing efficient and effective storage strategies for XML management systems. OrientX exploits schema information in the design and implementation of two storage strategies[8]: Clustering Element-Based (CEB), and Clustering Subtree-Based strategies. OrientX also implements the above schema-independent storage strategies: the strategy in Lore(We call it Depth-First Element-Based) and the strategy in Natix(We call it Depth-First Subtree-Based).

Depth-First Element-Based: In Depth-First Element-Based (DEB) strategy, each element node is a record, and records are stored in Depth-First order. This is quite the same to Lore.

Depth-First Subtree-Based: Depth-First Subtree-Based (DEB) strategy divides the XML document tree into subtrees according to the physical page size, such that each subtree is a record. Records are stored in depth-first order.

Clustering Element-Based: The Clustering Element-Based (CEB) storage strategy is similar to DEB as used in Lore and TIMBER in which each element node is a record. However, instead of storing the records in a depth-first traversal fashion, CEB clusters the element records such that records with the same TagName are placed close together.

Clustering Subtree-Based: The Clustering Subtree-Based (CSB) storage strategy first partitions the schema graph into semantic blocks. A semantic block describes a relatively integrated logical unit. We use the following heuristic to obtain semantic blocks: A node in a schema graph is the root of a semantic block if: a).it is a root of the schema graph, or b).it has a cardinality of '*' or '+', and it has children nodes. A record in the CSB storage strategy is an instance of a semantic block. All the instances of the same semantic block are clustered together.

Figure 2 shows an example of semantic blocks and records. (a) is the Schema graph, and the dot line partitions the graph into two semantic blocks: vendor (name, book) and book (publisher, title). (b) is an instance document, and the dot line implies three records here: the records b1 (p1, t1) and b2 (p2, t2) are instances of the semantic block book (publisher, title), while the record v(n, b1, b2) is an instance of vendor (name, book). b1 (p1, t1) and b2 (p2, t2) will be clustered which implies fewer I/Os for a query.

The storage strategy is chosen by the schema and the query requests. For instance, there must be few nodes having the same tag name in the documents having comparable size with the schema. Then cluster-based methods will lead to high percentage of free space in pages. So the DEB or DSB strategies can save more storage space. On the other hand, when the documents size is larger out than the schema, CSB and CEB methods may be more attractive. Another example, if there are many text nodes in the schema, then most of query requests are location. In that case, the navigation on depth-first methods will be more efficient. Otherwise, if there are many value nodes in the schema, and most of query requests are range condition, then the system will prefer cluster-based methods.

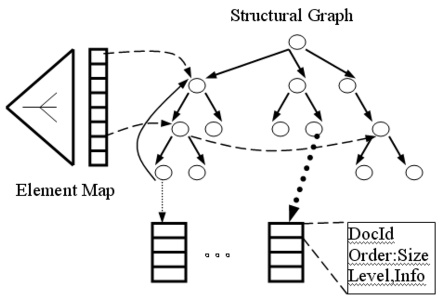


Figure 3: SUPEX Structure.

4. SUPEX: A SCHEMA-GUIDED PATH INDEX FOR XML DATA

An XML data set is a forest of rooted, ordered, labeled trees. Each node corresponding to an element and the edges represent elment-subelement relationships. A key issue in XML query processing is how to determine the ancestor-descendant relationship between any two elements. We adopt the numbering scheme proposed by [6]. Every node in XML document tree is associated with an *encode (DocIdOrder: SizeLevel)*. For any two given nodes *n1(d1, o1: s1, l1)* and *n2(d2, o2: s2, l2)* of a XML document tree, *n1* is an ancestor of *n2* if and only if *d1=d2*, *o1 <o2 <o1+s1*. This encoding scheme is applied to XML document tree and our index.

As shown in Figure 3, SUPEX[11] consists of two structures: a structural graph(SG), and a element map(EM). SG is constructed according to the schema, and represents the structure summary of XML data. EM provides fast entries to the nodes in SG. SG is a structural summary of top level elements in the schema. So all possible path starting from the roots of XML documents conforming to special schema will appear in SG. EM is a B+ tree. The key of EM is element name defined in the schema, and the entries in leaf nodes include a pointer to the nodes in SG. EM is useful for finding all elements with the same tag.

SG has one root node. Every node in SG except root node has a label which is a tag name defined in the schema, called E-Label. Except pointers to children nodes, every node has two pointers, one pointing to another node in SG which has the same E-Label called label pointer, another pointing to a set of fixed-length records of elements having the same incoming label path from the root of the document tree as itself called element pointer. We call the recoreds set the extent of the corresponding node.

Figure 3 shows the SUPEX structure, in which thin broken line means label pointer, wide broken line means element pointer and real line means children pointer.

SG is a tree when there is no cycle in the schema. When the schema is cyclic, SG is still like a tree by omitting the

```
<results>{
    for $b in doc("bib.xml")/bib/book
    where $b/price > 50
    return
        <result>
          { $b/title }
          { $b//author }
        </result>
}</results>
```

(a) XQuery Exprssion Example

```
Results
( for $b in ( doc("bib.xml")), child::bib, child::book)
  return
    Let tmp0:=
      ( some $tmp1 in ($b, child::price)
          satisfies
             return  $tmp1 > 50 )
    return
      if ( tmp0)
        then
          result(
              ( $b, child::title),
              ( $b, descendant::author)  )
        else ( )
```

(b) XAlgebra Expression

```
Results
( for $b in JOIN (name-index (bib), name-index(book)
  return
     Let tmp0:= ( ( $b, child::price) > 50 )
     return
       if ( tmp0)
         then
           result(
               ( $b, child::title),
               ( $b, descendant::author)  )
         else ( )
```

(C) Rewritten Query Plan

Figure 4: Query Processing Example.

edges pointing to ancestor nodes. To keep the equivalence between the schema and SG, a tag is used to mark the cycle. To facilitate the query processing, we associate each node with a encode like (Order: Size, Tag: BeginOrder). Tag can be one of three values: 0, 1, or 2. 0 indicates that the corresponding node is not in a cycle. 1 indicates that the node is in a cycle, and 2 means that the node is in a subtree rooted at a node of tag 1. Base on this encoding scheme, we can judge whether any two nodes in SG have the ancestor-descendant relationship.

SUPEX supports two basic queries: (1) given a tag, all elements with this tag can be obtained by the lookup of EM. (2) Simple label paths from the root of document can be matched by traversal of SG starting from the root.

In addition to these basic structural relationships, our index can support partial label path matching. For label paths like `‘//E1/E2/.../En’`, we needn't traverse the index graph to get result nodes. By the lookup of EM, we can obtain the head node of lists with E-Lable `E1`. For each node in this list, the subtree rooted at it will be traversed to find nodes matching ‘E1/E2/ . . . /E’. So only a part of SG will be traversed to get the result. This will greatly reduce the cost of partial label path matching.

5. XALGEBRA BASED QUERY EVALUATION

XML Query Enginee, the most important component in OrientX system, supports the Xquery1.0 recommended by W3C Work Group.The query engine does not cover all the features of XQuery[12] but captures its essence.

The Core syntax[1] defines the formal semantics of XQuery and provides a naive processing model. XAlgebra is similar to the XQuery formal semantics, but improves it in several aspects: *a)*Core syntax decomposes a path expression into a series of for-nest-loops, puts a distinct-operator after each evaluation of path expression. XAl gebra only decomposes a path expression onto a series of simple paths, which can be processed by matured path matching techniques, rather than naive nest-loop manner. *b)* Following Core syntax as the processing plan, nest-loop is the only choice for processing, and it is hard for Core syntax to take the advantages of storage, index, and optimization techniques on DBMS. Furthermore, it will show a inefficiency for evaluating a query on large XML documents. XAlgebra introduces some smart processing methods such as sort-merge join, predicate pushing-down, unnesting, etc, to improve the efficiency of processing. This will improve the efficiency a lot. *c)* Since OrientX is schema-dependent, the schema contains the informaiton of the document structure and the node type definition. These information can be helpful to improve the efficiency of type checking before query processing.

More details about XAlgebra is out of the scope of this paper. For the reason of space, we only propose the sketch of query evaluation with a simple example.

The query processing flow is as following. An input query is first parsed into a syntax tree and transformed into an internal expression represented in XAlgebra in the parser analyzer. The query in internal expression is then passed to the optimizer. In the optimizer, the original query plan is reorganized based on a set of rewriting rules and statistical information. The optimization result, a physical query plan is constructed and it will be evaluated by the evaluation engine through a set of call to the index manager and data manager. At last, the evaluation result, an XML tree may be offered to advance development through the OrientX API, or materialized to an XML document.

Figure 4(a)is an example of XQuery expression, which lists the title and authors, grouped by `result` element for each `book` with `price` larger than 100 in the `bibliography`. After parsing and analyzing, the query is transformed into the internal expression shown in Figure 4(b). Note that, each path expression in Figure 4(a) is decomposed as a list of simple paths.

Assuming name index for `bib` and `book` elements are available, then all `bib` and `book` elements can be retrieved by index; and a containment join is used to match the parent-child relationships among the `bib` and `book` elements. Note that only a document (`bib.xml`) is referred in the query, hence the simple path doc(‘bib.xml’) can be omitted here, for all the data to be accessed from the document root. Therefore the expression `‘for $b in ( doc(‘bib.xml’), child::bib, child::book)’` in Figure 4(b) can be rewritten as `‘for $ b in JION( name-index (bib), name-index (book) )’` . Assume that there is one and only one `price` element existing under each `book` element, then the some-quantify expression can be optimized as `‘($ b, child::price) > 50’`. After optimization, the query plan in figure 4(b) is rewritten as the one shown in figure 4(c).

6. HISTOPER: A HISTOGRAM BASED QUERY OPTIMIZER

HistOper supports cost based query optimization. HistOper

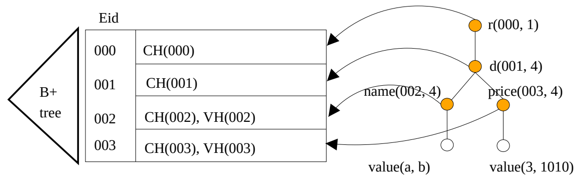


Figure 5: Statistical Information Model.

can estimate the cost of the paths or subpaths in the query, and find an optimized execution plan.

Selectivity estimation of path expressions in querying XML data plays an important role in query optimization. A path expression may contain multiple branches with predicates, each of which having its impact on the selectivity of the entire query. Previous methods of selectivity estimation have not taken into consideration the correlation between attribute values and their hierarchical positions, and assume instead that the selectivity of attribute values on different nodes and structures is independent and uniform.

HistOper builds an novel statistical information integrating the histograms into the schema information, which can capture the correlations in the XML data. Unlike most histogram method that process the structure and the value independently, we propose a novel method based on 2-dimensional value histograms to estimate the selectivity of path expressions embedded with predicates. The value histograms capture the correlation between the structures and the values in the XML data. Another type of histograms, coding histograms, keeps track of the position distribution. Figure 5 is an example of statistical information. Abstractly, our statistical information model is a node-labeled, directed graph structure $G=(N, E, D, R, L)$, where each node $n \in N$ is an element or a value. Each element has an identity *Eid*. Each edge $e \in E$ corresponds to the containment relationship between the nodes. D is a region maintains the count of the elements or the domain of the values. R is a recursive nesting tag marking if the node is in the recursive circle. Each element node has a link $l \in L$ points to the coding histograms of the node, if the node have value as its child, the *VH* are also exist. A B+ tree index is accepted to find the histograms with an Eid.

We define six basic operations on the value histograms as well as on the coding histograms: `V, S, D, A, PA,` and `PD`. The first two, `V` and `S`, are used to study value correlation, and the rest are for hierarchical correlation. We then construct a cost tree based on such operations. The selectivity of any node (or branch) in a path expression can be estimated by executing the cost tree. Figure 6 shows an example cost tree for the selectivity of `r` in the path `//r/d[price < 100]`. First, we can get the preliminary *coding histogram (SH)* by applying the `V` operation on the *VH* of the value node and the value domain of the predicate. Then we can use the `S` operation on the *SH* and the original *CH* of the value node to get the new *CH*, which only counts the nodes satisfying the predicate in each grid. At last, two `A` operations are used to get the selectivity of `r` in the path. The cost tree can be simplified by some rules if we know some nodes in the path are not self-nested through the schema information.

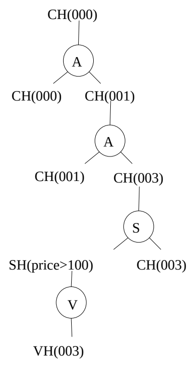
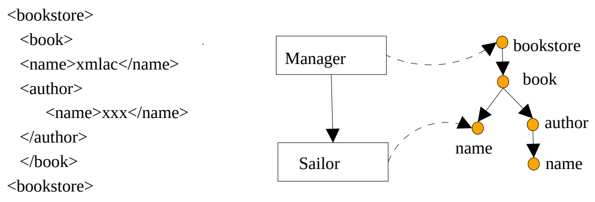


Figure 6: Cost Tree.



```
<bookstore>
  <book>
  <name>xmlac</name>
  <author>
      <name>xxx</name>
  </author>
  </book>
<bookstore>
```

Figure 7: Mapping Role to Node.

7. NODE-MAPPING ROLE BASED ACCESS CONTROL

Because of the different data model between relation data and XML data, the access control mechanism in relational database is not capable of managing XML data any more. Some important aspects need to be reconsidered, such as the granularity of access control, the semantic of authority, the relation among the rights on relative node in XML structure and so on. At the same time, large data capacity and alteration of the data also should be care of. RBAC has just been approved to be the member of ANSI. It makes access control powerful and simple. We extend the RBAC for OrientX, calling Node-Mapping Role Based Access Control.

We observed that role hierarchy and the XML data have some similarities. For example, there is an XML data section about a *bookstore*(Shown in Figure 7). If we define that *Manager* role can access all the information of this section and *Sailor* role can access just the *book name*, then we can map the *Manager* role to the node *bookstore* and the *Sailor* role to the node *name* in the schema of the data(Shown in Figure 7).

In formalization, if role A is the superior of role B, the data set which role A can access should be the superset of the data set which role B can access. Then we can map the two roles to two nodes in XML structure which has ancestor-descendant relation (ancestor is the superior and descendant is the junior). A role defined the access rules of a subtree of the Schema. In this way we can be awarded both the excellence of Role and the convenience of XML data access controlling. For example, we have used the Dynamic Separation of Duty Relation(DSD) characteristic to solve the problem of illegal association information accessing[3].

A role is a set of 3-uary tuples: `R = {` `Node, Context, Action` `}`,where `Node` denotes the tag name of the root of the subtree in XML document; `Context` is a path and defines the unique position of the node in the schema, and the `Action` represents a collection of allowed operation on the node, includeing read, insert, delete and update.

The roles can be awarded to the user in two ways: positive and negative. The positive roles assign the actions user can do, and the negative roles assign the actions user cannot do. Based on the compatibility, there are two kinds of roles: general roles and DSD roles. A user can choose many general roles during one session, and only one DSD role during one session. Introducing negative roles enable the user assigning the rules flexibly.

8. RELATED WORK

There are several Native XML Database systems, for example, Natix[5], Timber[4] and Tamino[9]. Natix is attractive of its storage model. XML documents are stored like the file trees in operating system, and the size of subtrees can be limited flexibly, XML data fragment insertion and deletion are also supported. But Natix does little support to XML schema. And its query performance maybe is inefficient, as it use D-Join step by step to match XPEs (also be limited with its index structure), which is time-consuming. Additional, Natix does not pay much attention to intermediate result and nested query translation, which are pivotal for XQuery implementation.

Timber is another mature Native XML Database, which brings forward the Pattern tree based XQuery processing method for the first time, and it proposed a proper logical algebra for XQuery. But its storage is borrowed from the Object-oriented database Shore, which reduces its query performance.

Tamino is a leading commercial native XML database, yet descriptions of its architecture are fairly sketchy.

Current native XML database systems are all schema-independented, so that they can not fully use schema to enhance the efficiency and ability of the system. OrientX is schema-based, and get the high overall performance of the system.

9. CONCLUSION AND DISCUSSION

In this paper, we described the system structure and design of OrientX, a schema-based native XML database proposed by Renmin University of China. Right now, the storage, query and index parts mentioned in this paper have already been implemented, and the query optimization, access control and update parts are under developing and will be finished soon. Next, we will publish OrientX to several high-level applications, such as E-government, news(base on NewsML), medication management and civil aviation system. During the system developing, we find some open problems for native XML databases:

- Is XQuery suitable for native XML database? Set-at-a-time processing method embodies the efficiency of traditional database. But it is hard to apply any set-at-a-time processing method on XQuery and its core syntax for its programming language style.

- Tree based algebra is a set-at-a-time method on XML data. But it is inadequate for expressing complex query request, especially for nesting query.

- Cost-based optimization method is a hot research topic for native XML database. But works on logical optimization seems few. A set of heuristic rules for accelerating the query is urgently needed. The availability of tradition rules such as predicate push down, need to be proved by a great deal of experiments.

- Without the semantic constraint in the schema, updating XML data will become unsafe. On the other hand, a strict schema will limit the flexibility of XML data. How to keep the balance?

After all, native XML database is a brand-new research field, and there are so many problems for us to deal with.

10. REFERENCES

[1] B. Choi, M. Fernandez and J. Simeon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. Technical Report MS-CIS-02-25, University of Pennsylvania, 2002.

[2] D. Florescu, D. Kossman (1999) Storing and querying XML data using an RDBMS. IEEE Data Eng Bull 22(3), 27-34.

[3] V. Gowadia, C. Farkas. RDF metadata for XML access control. In Proc. of 2003 ACM workshop on XML security. 39-48.

[4] H. V. Jagadish, Shurug AL-Khalifa, et al. TIMBER: A Native XML Database. Technical Report, University of Michigan, April 2002.

[5] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML data. In Proc. of 16th ICDE. San Diego, California, USA, February 2000. 198.

[6] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions, In Proc. of 27th VLDB,Roma,Italy, 2001

[7] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, Vol.26(3):54-66, September 1997.

[8] X. Meng, D. Luo, M. Lee and J. An. OrientStore: A Schema Based Native XML Storage System. In: Proc. of VLDB 2003. Berlin, Germany. 1057-1060

[9] H. Schoning. Tamino - A DBMS Designed for XML. In Proc. of 17th ICDE. Heidelberg, Germany, April 2001. 149-154.

[10] I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.s. Weld. Updating XML. In Proc. of 2001 SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, May 2001.

[11] J. Wang, X. Meng and S. Wang. SUPEX: A Schema-Guided Path Index for XML Data. Doctoral Poster. In: Proc. of 28th VLDB Conf. Hong Kong, 2002.

[12] XQuery1.0: An XML Query Language,W3C Working Draft,April 2002.Available At http://www.w3.org/TR/xquery/

[13] Apache Xindice. http://XML.apache.org/xindice/