Processing XML Queries on Structure and Keyword in SKEYRUS

Dao Dinh Kha †, Masatoshi Yoshikawa †‡, and Shunsuke Uemura †

† Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan

‡ Information Technology Center, Nagoya University
Furo-cho,Chikusa-ku, Nagoya 464-8601, Japan

Abstract

Applying numbering schemes to simulate the structure of XML data is a promising technique for XML query processing. In this paper, we describe SKEYRUS, a system, which enables the integrated structure-keyword searches on XML data using the rUID numbering scheme. rUID has been designed to be robust in structural update and applicable to arbitrarily large XML documents. SKEYRUS accepts XPath expressions containing word-containment predicates as the input, therefore the query expressiveness is significantly extended. The structural feature and the ability to generate XPath axes of rUID are exploited in query processing. Preliminary performance results of SKEYRUS were also reported.

1 Introduction

Presently, Extensive Markup Language (XML) [12] has become a framework for structural information exchange over the Internet. Processing XML data requires new techniques different from the techniques applied in relational databases, the structure of which is fixed by pre-defined schemes. XML elements must be assigned unique identifiers in order to be distinguished from each other. In addition, the structure of XML documents may change when elements are inserted or removed. Therefore, an effective mechanism, which can not only generate robust element identifiers but also maintain the information about the structure of XML documents, is essential for processing XML queries. Using a numbering scheme, which generates the identifiers of XML elements in such manner that the hierarchical order within XML documents can be re-established based on the element identifiers, can meet the requirement.

Several numbering schemes for XML data have been proposed in [2, 5, 11, 10]. Among these schemes, the *Unique Identifier* (UID) technique that uses a k-ary tree to enumerate nodes in an XML tree has an interesting property whereby the identifier of a parent node can be determined based on the identifiers of its child nodes. Given a node having the identifier i, the identifier of its parent node can be computed using the formula $\lfloor (i-2)/k+1 \rfloor$. This property is promising in evaluating the structural part of XML queries because it enables an effective re-establishment of hierarchical order within a tree.

However, the UID technique is not robust in structural update. When a node is inserted, the identifiers of the sibling nodes to the right of the inserted node and their descendant nodes will be also changed. If a node insertion causes the fan-out of a node be larger than the pre-defined value k, the entire identifier system has to be recomputed. Furthermore, the value of UID easily exceeds the maximal manageable integer value. Therefore, an additional library as well as an extra integration cost is required.

In [3], the *recursive UID* numbering scheme has been proposed so as to remove the above-mentioned drawbacks. Besides the property that the identifier of a parent node can be determined based on the identifiers of its child nodes, rUID is robust for structural update and applicable to arbitrarily large XML documents. In addition, the structural feature of rUID is effective for XPath axes processing.

In this study, we describe SKEYRUS (*S*tructure and *KEY*word search based on *R*ecursive *U*id *S*ystem), that integrates structure and keyword searches on XML data using rUID. SKEYRUS is comprised of the modules for raw data processing and managing, query plan generation, content query processing, and integration. The input of SKEYRUS is a simplified XPath expression with word-containment predicates. Taking into account the feature of query processing using numbering schemes such that a query plan normally contains many joins and a number of joins may be repeated in various queries, we have discussed the following issues:

- *A join mechanism.* How to exploit rUID to perform joins effectively?
- *Query plan selection.* Which is the appropriate plan to execute a query?
- *Physical data organization.* How to partition the data into files appropriate to the query processing using rUID?
- *Dealing with the common subqueries.* How to keep the result of the frequent subqueries to save the repeatedly processing cost?

The rest of paper is organized as follows. Section 2 briefly reviews related works. Section 3 presents an overview of the 2-level recursive UID that is the core data structure in SKEYRUS. Section 4 describes the design of SKEYRUS. Section 5 discusses a number of preliminary performance results of SKEYRUS. Section 6 concludes this paper with future work suggestion.

2 Related works

Numbering scheme proposals. A method to determine the ancestor and descendant relationship using *preorder* and *postorder traversals* has been introduced in [2]. Extensions of the method using the *preorder* and *range* have been presented in [11, 1]. Another approach uses the *position* and *depth* of a tree node to process containment queries [16]. The application of XID-map in change management of versions of XML documents have been discussed in [5].

The UID technique has been introduced in [10]. Applications of the technique have been described in [6, 7]. In these studies, the problems of structural update and the overflow of identifier have not been discussed.

XML data management approaches. Respecting the role of RDBMS, there are several approaches to implement a system to manage XML data, as follows:

- *All-In*: The system is implemented exclusively based on an RDBMS.
- *All-Out*: The system is implemented without using any RDBMS. Even the indexing and storage modules are implemented originally.
- *Half-In-Half-Out*: An RDBMS is used to store the pre-processed data. The query processing is performed by other modules, which are built originally, taking into account the specific requirements and data models.

In the *All-In* approach, data is stored entirely in an RDBMS. The queries on XML data are re-written in SQL statements and are performed using the built-in SQL module of the RDBMS. There is a number of works applying the *All-In* approach, [4, 9, 15]. The transformation of queries from XPath to SQL statements has been described in XRel, [15]. Integration of keyword search to XML query processing has been discussed in [4]. The maturity of the relational database technology and the plenty of RDBMS products are the advantage of the *All-In* approach. The disadvantage is that RDBMS have been designed primarily for the relational data model, not for XML, which also includes the structural information of data.

The *All-Out* approach is applied for *native* XML databases. The module used to store and index data is implemented usually using B^+ tree. The query processing is performed based on the techniques specially designed for XML data, such as structure summary or numbering scheme. The recent examples of the *All-Out* approach have been described in [6, 11]. The advantage of the *All-Out* approach is the fast performance. On the other hand, the *All-In* approach usually requires an implementation workload heavier than the first approach does.

In the implementation of SKEYRUS, we have adopted the *Half-In-Half-Out* approach. An RDBMS is used to perform the task of managing the pre-processed data. Other modules in SKEYRUS have been implemented originally. The purpose of the approach adoption is to save the workload and concentrate to the new design. For a reliable evaluation, we have measured the time necessary for each task, such as data loading and processing, separately. There no obstacle to transform our system from the third approach to the first or second approaches when it is needed.

3 Overview of 2-level rUID

As mentioned above, the rUID numbering scheme is crucial for SKEYRUS. In this section, we describe the main features of 2-level rUID, the simplest version of the multilevel recursive UID, in order to make the paper self-contained. The technical details of rUID has been presented in [3].

3.1 Description

The 2-level rUID numbering scheme manages the identifiers of nodes in XML trees by the *global* and *local* levels. The set of nodes is divided into subsets, the identifiers of which are created in the *global* level whereas the nodes of each subset are managed in the *local* level. A number of parameters are generated to be used in both of these levels. The additional data is small enough to be comfortably loaded into main memory allowing fast access when navigating inside the XML tree.

The construction of the 2-level rUID numbering scheme for an XML tree consists of the following steps: (1) partition the XML tree into areas, each of which is an induced subtree of the XML tree; (2) enumeration the newly created areas according to the original UID scheme in order to generate the global indices; (3) for each area, enumeration of the nodes of that area in order to generate the local indices; and (4) compose of the synthetic identifiers of nodes from the global and local indices. The 2-level rUID is formalized by the following three definitions:

Definition 1 *(A frame) Given an XML tree $\mathcal{T}$ rooted at $\boldsymbol{r}$, a frame $\mathcal{F}$ is a tree: (1) rooted at $\boldsymbol{r}$, (2) the node set of which is a subset of the node set of $\mathcal{T}$ and (3) for any two nodes $\boldsymbol{u}$ and $\boldsymbol{v}$ in the frame, an edge exists connecting the nodes if and only if one of the nodes is an ancestor of the other in $\mathcal{T}$ and there is no other node $\boldsymbol{x}$ that lies between $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathcal{T}$ and $\boldsymbol{x}$ belongs to the frame.*

Definition 2 *(UID-local area) Given an XML tree $\mathcal{T}$ rooted at $\boldsymbol{r}$, a frame $\mathcal{F}$ of $\mathcal{T}$, and a node $\boldsymbol{n}$ of $\mathcal{F}$, a UID-local area of $\boldsymbol{n}$ is an induced subtree of $\mathcal{T}$ rooted at $\boldsymbol{n}$ such that each of the subtree's node paths is terminated either by a child node of $\boldsymbol{n}$ in $\mathcal{F}$ or a leaf node of $\mathcal{T}$, if between the leaf node and $\boldsymbol{n}$ in $\mathcal{T}$ there exists no other node that belongs to $\mathcal{F}$.*

A frame divides an XML tree into local areas, each of these areas is rooted at a frame node. From the definition 2, the intersection of any two areas is either empty or consists of only one node, which is the root node of an area and also a leaf node of the other area. Hereafter, let us refer to the full identifier of a node as its *identifier* and the number assigned to a node locally inside an UID-local area as its *index*. Let κ denote the maximal fan-out of nodes in $\mathcal{F}$. We use a κ-ary tree to enumerate the nodes of $\mathcal{F}$ and let the number assigned to each node in $\mathcal{F}$ be the index of the UID-local area rooted at the node. The 2-level UID numbering scheme based on $\mathcal{F}$ is defined as follows:

Definition 3 *(2-level rUID) The full 2-level rUID of a node $\boldsymbol{n}$ is a triple $(\boldsymbol{g}, \boldsymbol{l}, \boldsymbol{r})$, where $\boldsymbol{g}$, $\boldsymbol{l}$, and $\boldsymbol{r}$ are called the global index, local index, and root indicator, respectively. If $\boldsymbol{n}$ is a non-root node, then $\boldsymbol{g}$ is the index of the UID-local area containing $\boldsymbol{n}$, $\boldsymbol{l}$ is the index of $\boldsymbol{n}$ inside the area, and $\boldsymbol{r}$ is $\boldsymbol{false}$. If $\boldsymbol{n}$ is the root node of an UID-local area then $\boldsymbol{g}$ is the index of the area, $\boldsymbol{l}$ is the index of $\boldsymbol{n}$ as a leaf node in the upper UID-local area, and $\boldsymbol{r}$ is $\boldsymbol{true}$. The identifier of the root of the main XML tree is (1, 1, $\boldsymbol{true}$).*

The rUID of a node in an XML tree is determined uniquely. We construct a table $\mathcal{K}$ having three columns *global index*, *local index*, and *fan-out*. Each row of $\mathcal{K}$ corresponds to an UID-local area and contains the global index of the area, the index of the area's root in the upper area, and the maximal fan-out of nodes in the corresponding area, respectively. The table $\mathcal{K}$ is sorted by the global index. The value κ and the table $\mathcal{K}$ are the *global parameters*, which are loaded into main memory dur:

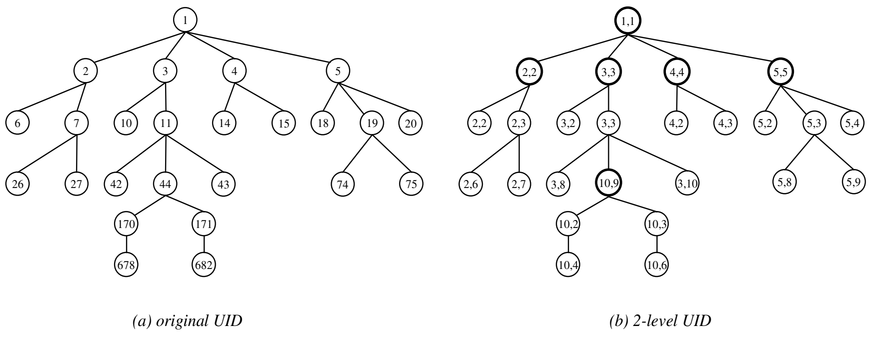

Figure 1. An original UID and its corresponding 2-level rUID version

Example 1 *Figure 1 depicts examples of the original UID and the new 2-level rUID. In the tree shown on the left, the number inside each node is the node's original UID. In the tree shown on the right, integers inside each node are the global and local indices of the node's 2-level rUID. Rather than showing root indicators, the root nodes are encircled by bold circles and the non-root nodes are encircled by fine-lined circles. Using the original UID, the value k is 4 and the maximal identifier equals 682. Using the rUID, the global fan-out κ is 4 and six UID-local areas exists. The table $\mathcal{K}$ of the global parameters of 2-level rUID is shown in Figure 2.*

Global index	Local index	Local fan-out
1	1	4
2	2	2
3	3	3
4	4	2
5	5	3
10	9	2

Figure 2. Global parameter table for the 2-level rUID shown in Figure 1(b)

3.2 Properties of rUID.

The rUID preserves the ability to compute the parent-child relationship. The algorithm shown in Figure 3 computes the rUID of a parent node. Besides that, rUID has the following properties:

```
Input:  An XML tree 𝒯, its κ and 𝒦, and the 2-level rUID of a node n
Output: The 2-level rUID (g,l,r) of the parent node of n

1.   Determine g as the index of the local area that n belongs to, not as the root
2.   Get the fan-out of the local area in 𝒦
3.   Compute the local index l
4.   If l equals 1 then refer to 𝒦 to get the correct l; Set r true
5.   If not, set r false
e.   Return (g, l, r)
```

Figure 3. Outline of the algorithm used to determine the parent node of a node

Scalability. rUID is applicable for arbitrarily large trees. In SKEYRUS, 2-level rUID is sufficient for generating the element identifiers of large XML data sets.

Robustness in structural update. In the original UID, if a new node is inserted into an XML tree when space is available then the insertion causes the identifers of the sibling nodes to the right of the inserted node as well as those of their descendant nodes, to be modified. In the worst case, when the insertion increases the tree's maximal fan-out, the entire enumeration has to be performed again. Identifiers of all of the nodes must be changed, which leads to an expensive reconstruction. Using rUID, the scope of identifier update due to a node insertion or deletion is reduced by a magnitude of two. For example, if a node is inserted, only the nodes in the UID-local area where the update occurs need to be considered. If an appropriate space is available for the new node, then among the descendants of the sibling nodes to the right of the inserted node, only those which belong to the same UID-local area will have their identifiers modified. The nodes in the descendant areas are not affected because the frame $\mathcal{F}$ is unchanged. Otherwise, if such a space does not exist for the newly inserted node then the fan-out of the tree used in enumerating the UID-local area must be enlarged. Rather than modifying the identifiers of every XML component, the enlargement changes only the identifiers of the nodes in this area. In both cases, since the size of an UID-local area is much smaller than the size of the entire data set tree, the scope of the identifier update is greatly reduced. The rUID deals with another structural operation, for example node deletion in a similar manner.

Hierarchical summary in the frame. The global index of rUID can be used to determine the relative position, such as preceding or following, of two nodes in the entire data tree, if these indices are distinguished. For example, if a node precedes (or follows, respectively) another node in the frame of an XML tree, the former precedes (or follows, respectively) the latter in the entire tree.

XPath axes expressiveness. XPath [13] is a language for addressing parts of XML documents. An important type of XPath expression is the *location path* that is a sequence of *location steps*. A location step has three parts: 1) an axis, 2) a node test, and 3) zero or more predicates. An initial node-set is generated from the axis and node test, and then filtered by each of the predicates in turn. A predicate filters a node-set with respect to an axis to produce a new node-set. Generating the XPath axes is essential in evaluation of location steps in XPath expressions. XPath axes can be constructed using rUID, in a manner similar to the algorithm in Figure 3 (see [3] for details). This property is used in the join mechanism of SKEYRUS. In some cases, the set of nodes generated is a superset of the considered axis but the extra does affect much on the speed of query processing in main memory.

3.3 Shortcut for Common Subqueries (SCSQ)

This technique aims for saving the processing cost of the *common parts* of queries. The issue is typical of the implementation of not only rUID but also other numbering schemes. In any query processing using numbering schemes, a number of joins may be frequently repeated in various queries. Let us consider an illustration example. Suppose that we have to process two queries "`//a/c/d/e/f[predicate-1]`" and "`//b/c/d/e/g/h[predicate-2]`". A naive solution is to process each of queries separately. However, these queries have a common subquery "`c/d/e/`", which indicates that the elements `c` and `e` participate in a relationship: "There exists an element `d` such that `c` is the parent of `d`, and `d` is the parent of `e`". If all pairs (`c`, `e`), `c` and `e` of which observer the relationship, are pre-recorded in the processing of the first query then the common subquery "`c/d/e/`" has not be performed again in the the processing of the second query. Hence, we store the result data of frequent common subqueries for late uses in order to avoid performing the repeated tasks in query processing. The data structure for the SCSQ is described in Section 4.2.

4 SKEYRUS - an application of the 2-level rUID

In this section, we describe an implementation of rUID in a prototype system named SKEYRUS. The system integrates two functions together: querying on structure of XML documents and searching by keywords. The input of SKEYRUS is expressed as a simplified XPath expression, predicates of which are word-containments. The output is the portion of XML documents that correspond to the query.

4.1 Approach

In SKEYRUS, we valuate XML queries by selecting the tested nodes that satisfy conditions on content, and then joining these qualified nodes by axes. The input queries are expressed by XPath expressions having word-containment predicates. The XPath axis construction ability of rUID is useful in processing the structural part of the queries. For the content part, we focus on keyword search, which is a popular request to the Web applications. For example, most of user's searches in the Internet aim at retrieving from a large data collection the data items containing given keywords. The keyword search is embedded in the predicates of XPath expressions by the function **contains**().

4.2 System design

SKEYRUS has the following components: a module to process raw XML data, a module to store and index the processed data, and a query processing engine. The engine consists of a query plan generator, a content processing module, an optimizer, and an integrator. The user interface of SKEYRUS is omitted here. The system design of SKEYRUS and the components are depicted in Figure 4.

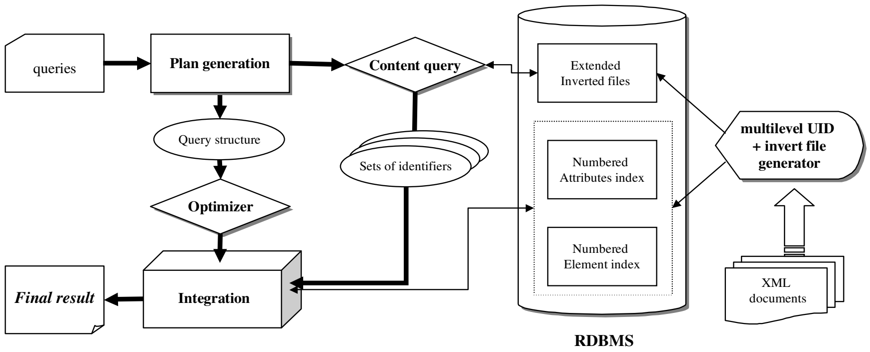

Figure 4. System design of SKEYRUS

Pre-processing, storing, and indexing. The input XML documents are parsed by the pre-processing module, and then the resulted XML tree's nodes are numbered using rUID. The global parameters κ and $\mathcal{K}$ are generated and stored in the table $globalParameters$.

In order to generate the structural information, for each XML element and attribute, a tuple `<rUID, EleAttrName, descr>` is generated. The `rUID` is the identifier of the element or attribute `EleAttrName`, and `descr` contains the type of `EleAttrName` and its position, which is the address where data resides, in XML documents. Two indices are created on `rUID` and `EleAttrName`. The structural information is saved to the table $codeName$.

For each leaf node, an inverted structure of the node content is created. The classic structure for full-text indexing is an inverted file that records the occurrence of words in documents by the pairs `<word, document>`. In [4], an extended structure has been proposed to record the occurrence of a *word* at a *depth* in a *location* of an *element* or *attribute* in XML documents. In SKEYRUS, since the hierarchical level can be computed using rUID, we can discard the *depth* parameter. Therefore, the structure of the rUID-keyword inverted file is `<rUID, word, descr>`. The ability of rUID to efficiently determine the ancestor-descendant relationship allows the BUS approach, [6], to be applied, by recording the word occurrences at only leaf nodes. The approach reduces the number of records needed to store the word occurrences. The content information is saved to the table $codeWord$.

The schemes of $codeName$ and $codeWord$ and several tuples are shown in Tables 1 and 2 (the attribute `descr` is omitted). The keys of these tables are (`gUID`, `lUID`, `rIndicator`) and (`gUID`, `lUID`, `rIndicator`, `word`), respectively. We use an RDBMS to store and index these tables.

Plan generation in SKEYRUS. An input query is expressed by an XPath location path. A

gUID	lUID	rIndicator	Name
int	int	char	char(30)
1	1	1	'personnel'
1	2	0	'person'
...	...	...	...

Table 1. Tables $codeName$: Element and attribute serialization

gUID	lUID	rIndicator	word
int	int	char	char(30)
1	20	0	'employee'
1	342	0	'nara.ac.jp'
1	343	0	'biology'
...	...	...	...

Table 2. A table $codeWord$: Words serialization

location path has the form: $\partial\ Step_1\ \tau_1\ Step_2\ \tau_2 \cdots Step_l$ where $l \geq 0$, ∂ is either an empty symbol or '/', τ_i $(i = 1..l-1)$ is '/', and Step$_i$ $(i = 1..l)$ is a *location step*. An initial *query plan* is generated by parsing the input location path into location steps. The location step are classified into two types:

1. the location steps, the predicates of which are either empty, for example "`child::doc`", or can be evaluated based only on the tested nodes in the location steps, for example "descendant::figure[contain(child::title,'`genetic`')]"
2. the location steps, the predicates of which are related with not only tested nodes but also the position of the tested nodes in the axis, for example "`child::chapter[position()=5]`"

A query plan is stored in a table having the attributes `stepID`, `nodeType1`, `axisType`, `nodeType2`, `Operation`, `OpType`, and `Output`. Each row of the table is a *plan step*. The `stepID` indicates the order to perform plan steps. The `nodeType1` and `nodeType2` show candidate node sets joined by the axis `axisType`. The `Operation` is the function executed on the candidate node sets to get the qualified nodes. The `OpType` is the type of location step. The `Output` indicates whether the location step produces the output.

Example 2 *A query plan of the XPath expression "`/child::doc/child::chapter [position()=5]/descendant::figure[contain(child::title,'genetic')]`", which selects the figure, the title of which contains the keyword 'genetic', of the fifth chapter of a document, is shown in Table 3.*

stepID	nodeType1	axisType	nodeType2	Operation	OpType	Output
1	root	child	doc			
2	doc	child	chapter	position()= 5	2	
3	chapter	descendant	figure			yes
4	figure	child	title	contains ‘genetic’	1	

Table 3. A query plan for Example 2

Shortcut for common subqueries. Common subqueries are extracted from the input. The intermediate result of common subqueries is stored in a table named `scomsubqTab` having three attributes `comquery`, `startnode`, and `endnode`, where `comquery` stores the expression of the subquery, `startnode` and `endnode` are the nodes, the relationship of which corresponds to the subquery. Each common subquery corresponds to a number of tuples.

If the number of occurrences of a subquery in the input expressions exceeds a threshold then the tuples corresponding to the subquery are generated. This generation is a query execution itself, the result of which is stored in `scomsubqTab` for reference. If a new query contains a subquery that has been already recorded then rather than performing the subquery again, the `startnode` and `endnode` corresponding to the subquery are used.

Content processing. We evaluate a location step by choosing the candidates, and then projecting these candidates into an appropriate axis. The candidates are generated by the content processing module that uses the inverted files created by the pre-processing module. The content processing module performs two tasks: Loading the list of elements or attributes having a given name, and filtering a list of elements or attributes by given keywords.

For example, the predicate (*contains*: ‘database’) is evaluated by checking the tables $codeWord$ and selecting the tuples, the attribute `word` of which is ‘database’. In the current version of SKEYRUS, the content processing is performed using SQL built in RDBMS. The SQL statement for the predicate is:

```
SELECT * FROM codeWord
WHERE word = 'database'
ORDER BY  gUID ASC, lUID ASC, rIndicator ASC
```

The output of the content processing module is stored in buffer tables in main memory. SKEYRUS manages the buffers using flags and frees a buffer after the buffer become not necessary. For example, to process the query plan in the example 2, the first buffer stores the `doc` elements. The second buffer stores the qualified `doc` elements that are children of the root. The first buffer now is freed and will be used to store the `chapter` elements. The result of the join of the first and second buffers on the paren-child axis is stored in the third buffer. The first and second buffers become free, and so on.

Structure processing. This module joins, using rUID as the key, the tables in main memory according to given axes. The output is the pairs of items from these tables. A pair is qualified if its components have a hierarchical relationship corresponding to the axis.

Notation. Let A, $\mathcal{C}$(A), $\mathcal{X}$(A), and $\mathcal{X}$(A, B) denote a list of elements of the type a, the list of items in A satisfying the condition $\mathcal{C}$, the computation of the axis $\mathcal{X}$ from A, and the join of the sorted tables A and B by the axis $\mathcal{X}$, respectively.

Join mechanism. To establish $\mathcal{X}$(A, B), rather than checking all possible pairs from tables A and B, we first generate $\mathcal{X}$(A) and store the finding in an buffer table E. The ability of rUID to compute XPath axes is crucial for this step. The axis mapping is performed in main memory without any disk I/O. The cost of this step is denoted by $cost(\mathcal{X}(\mathrm{A}))$. Next, we sort E by rUID and store in a table E'. The cost of this step is $O(|E| * log|E|)$. Finally, we merge E' with B. Because E' and B are sorted, the cost of this step is $(|E'| + |B|)$ or $(|E| + |B|)$. The last two steps is similar with *Sort-Merge Joint* in RDBMS. The total cost of the join equals $cost(\mathcal{X}(\mathrm{A})) + O(|E| * log|E|) + (|E| + |B|)$. The join process is depicted in Figure 5.

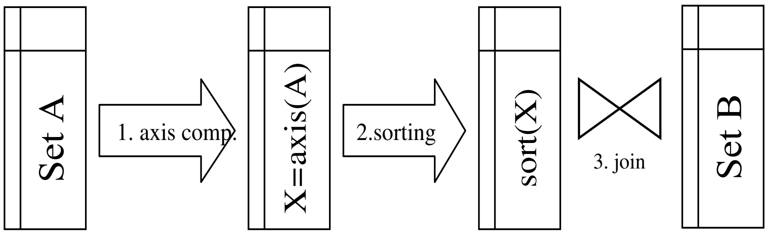

Figure 5. Join process in SKEYRUS

This join mechanism can be applied for different axis types. For sorting, the hierarchical order is used to decide whether two nodes are swapped. For example, the following proposition can be used in the preceding and following axes joins.

Proposition 1 *Given three nodes* $\boldsymbol{p}$, $\boldsymbol{n_1}$, *and* $\boldsymbol{n_2}$, *if* $\boldsymbol{p}$ *is a preceding node of* $\boldsymbol{n_1}$ *and* $\boldsymbol{n_1}$ *is an ancestor or preceding node of* $\boldsymbol{n_2}$ *then* $\boldsymbol{p}$ *also is a preceding node of* $\boldsymbol{n_2}$. *Similarly, if* $\boldsymbol{p}$ *is a following node of* $\boldsymbol{n_1}$ *and* $\boldsymbol{n_1}$ *is a descendant or following node of* $\boldsymbol{n_2}$ *then either* $\boldsymbol{n_2}$ *is the root node or* $\boldsymbol{p}$ *is a following node of* $\boldsymbol{n_2}$.

Rules for query plan selection in SKEYRUS. The general query optimization for XML data has been discussed in [8]. In this part, we discuss several improvement measures specific to query processing on numbering schemes.

Given a query plan, the query steps can be performed in different orders. For example, for the query plan in Table 3, the steps can be performed in the order 1, 2, 4, and then 3. Another order is 1, 2, 3, and then 4. The different process orders require the different query processing time and main memory for buffering. The aim of our optimization is to choose the process order appropriate to our criterion. Regarding the *input/output* and *processing time consumption*, we consider the following illustration example.

Example 3 *Consider a simple query "*`a/`$\mathcal{X}$`::b[satisfies C]`*" where* $\mathcal{X}$ *is an axis,* $\mathcal{C}$ *is a condition on* `b`. *The process order is either generating the set of nodes* `b` *satisfying* $\mathcal{C}$ *and checking which* *nodes belong to the axis $\mathcal{X}$ of* `a`*, or joining* `a` *and* `b` *by the axis $\mathcal{X}$ first, and then checking which nodes satisfy $\mathcal{C}$. Let P_1 and P_2 denote these plans, which are shown in Table 4.*

stepID	Plan P_1	Plan P_2
1	find $\mathcal{C}$(B)	find A
2	find A	find B
3	find $\mathcal{X}$(A, $\mathcal{C}$(B))	find C = $\mathcal{X}$(A, B)
4		find $\mathcal{C}$(C)

Table 4. Two query plans for Example 3

The cost difference between P_1 and P_2 is d = cost(P_1) - cost(P_2), or cost($\mathcal{C}$(B)) - cost(B) - cost($\mathcal{C}$(C)) + cost($\mathcal{X}$(A, $\mathcal{C}$(B))) - cost($\mathcal{X}$(A, B)). From the cost of the join mechanism, d is found equal to cost($\mathcal{C}$(B)) - cost(B) - cost($\mathcal{C}$(C)) + $|\mathcal{C}(B)|$ - $|B|$. The value depends on the cost of evaluating $\mathcal{C}$ and the size of C.

We adopted the following approximate measures to select the query plans:

- *Number of connections between processing modules in main memory and RDBMS*: The smaller the number of connections between these modules, the better a plan is. We must cluster the query steps performed by RDBMS in batches. The numbers of the connections in P_1 and P_2 are two and three, respectively. In term of this measure, P_1 is better than P_2.
- *Cost of the evaluation of $\mathcal{C}$*: If the cost is *cheap* then P_1 is better than P_2. If the cost to verify $\mathcal{C}$ is *expensive* then P_2 is better than P_1. For simplicity, we consider the number of input keywords in $\mathcal{C}$ the cost of $\mathcal{C}$.

Regarding *memory consumption*, we choose the number of buffers used to store the intermediated data as a measure. Intuitively, a buffer can be clear only if it is not referred by latter joins. Therefore, keeping the number of buffers, which are recursively referred, to the smallest value is desired. In order to keeps the number of main memory buffers small, the order to process the query steps is from top and bottom steps toward the step that produces the output.

Database table tuning. The number of tuples used to store the information structure is equal to the number of elements and attributes. However, the number of tuples recording the occurrences of words in the content of elements and attributes is larger. For example, the 3172 Kb data set II described in Section 5 needs 151172 tuples to record the occurrences of words. When the number of tuples is too large, distributing the tuples into smaller tables is necessary to speed up the query processing. However, the distribution raises the question of how to choose the correct tables to make the I/O operations.

Creating the name of tables from the two parts is our solution to the problem. For the tables of $codeName$ type, the first part of the name of a table is a common prefix of element names stored in the table. For the tables of $codeWord$ type, the first part of the name of a table is the common prefix of words stored in the table. The second part, in both of these cases, is the common global index of rUID. For example, the table named “table.go.162” stores the items starting by the characters ‘GO’, ‘Go’, ‘gO’, or ‘go’, the global index of which equals 162. The XPath axis expressiveness of rUID can be used in order to choose the most appropriate tables to apply SQL statements. For example, the processing of the query “`a/axis-`$\mathcal{X}$`::b`” can be: load A, compute $\mathcal{X}$(A) as C, using C to select the tables to load B.

5 Experiment

In this section, we describe several experiments on SKEYRUS. The test results demonstrates the scalability and the effectiveness of rUID in querying XML data.

Experimental platform. The experiments were conducted in a workstation running on Windows XP Professional with two dual CPU 2GHz, 2Gb of RAM and a hard disk to store data.
Software utilities. The RDBMS is Oracle 9*i* Personal Edition. The DOM parser available from the Xerces project (version 1.0.3) of Apache Software Foundation [14] is used to parse XML data. All other test programs were written in Java. The connection of the test programs with the RDBMS is performed through JDBC-ODBC.

Data sets. We used the Shakespeare’s plays formatted by XML available at http://sunsite.unc.edu/pub/sun-info/xml/eg/shakespeare.1.10.xml.zip. In addition, we generated two synthetic data sets that supposedly describe the personnel organization, denoted by `personnel`, of a company. The main entity is `person`, which has a profile including the attributes `identifier`, `contract status`, `name` (`family` and `given`), `email`, and technical `link`. Each person may manage a number of subordinates and may be managed by a manager. The specification of the synthetic data sets is shown in Table 5.

Comparison of the capacities of UID and rUID. The UID technique failed to numerate entirely the data set I, which has an unbalanced structure and a high degree of recursion. The UID technique can deal with Shakespeare’s plays because the documents have a relatively balanced and low tree shape. However, when the plays are grouped in a *collection*, for example by adding a DTD declaration `<!ELEMENT collection (play+)>`, then the UID technique was overflowed, even when only a single play “Love’s Labor’s Los” was added. Meanwhile, the 2-level UID is successful enumerated both of these data sets. The number of UID-local areas in the first case and the second case is 131 and 154, respectively.

rUID and XML query processing. We compared the query processing of SKEYRUS with Xalan [14] on the data set II, which is much larger than data set I. Xalan is a shareware for XML data processing that supports XPath expressions, including the axes. Hence, we could test the effectiveness of the XPath axis expressiveness of rUID in query processing. We conducted the experiments using the query input in the following categories:

- Q1: simple XPath expressions having parent and ancestor axes, such as “`//child::person`” and “`//child::person/child::email`”.

- Q2: XPath expressions having parent and ancestor axes that require the hierarchical level information, such as
"`//child::person/child::*/child::person`" and
"`//child::person/child::*/child::*/child::person`".

- Q3: complex XPath expressions having keyword search, such as
"`/descendant::person/child::note[contains(self,'academic')]`".

- Q4: XPath expressions having preceding axes, such as
"`/descendant::person/preceding::person`".

Let x_p and x_q, s_p and s_q denote the times for data preparing and querying of Xalan and SKEYRUS, respectively. For Xalan, x_p is the time for reading the data file and parsing it. For SKEYRUS, s_p is the time for loading needed data from RDBMS. The times for processing these queries are shown in Table 6. Each test consisted of three runs and the data of the most shock run was discarded. The time average of the two other runs was considered as the result of the test.

Data set	Size	# of ele./attr.	tree height	description
Synthetic set I	7.6Kb	201	11	20 main entities
Love's Labor's Lost	210Kb	5057	7	plays grouped in `collection`
Synthetic set II	3172Kb	50052	11	7104 main entities

Table 5. Speci£cation of the data sets

Categories	x_p	x_q	x_p+x_q	s_p	s_q	s_p+s_q
Q1.1	860	1312	2172	1344	–	1344
Q1.2	875	1562	2437	2500	865	3365
Q2.1	875	1750	2625	2453	991	3444
Q2.2	875	1875	2750	2500	1084	3584
Q3.	891	1328	2219	2844	20	3044
Q4.	875	X	–	2650	2050	4700

Table 6. Times of query processing (ms)

Analysis. From the Table 6, we have made the following observations:

- Accessing RDBMS to retrieve the pre-processed data in SKEYRUS requires more time than parsing a text file in Xalan. This can be best explained that SKEYRUS has to communicate with the RDBMS via ODBC-JDBC.

- When data is loaded into main memory, SKEYRUS is better than Xalan in query processing. In average, the processing time required by SKEYRUS is 50-60% of the time required by Xalan.

- When processing an axis, if the names of the participated elements are known, then the scope of matching is limited to the data of only these elements. If one of element participants in an axis is unknown, as in Q2.1 and Q2.2, considering all of possibilities of the component is necessary. Therefore the required time is increased. SKEYRUS deals with the cases using the ability of rUID for hierarchical level determination, so the required time for processing was not increased sharply.

- We tested two query plans for Q3. The first plan is {(load `word`='academic' to the buffer 1), (load `name`='note' to the buffer 2), (join the buffers 1 and 2), (load `name`='person' to the buffer 3), (join the buffers 2 and 3 by the *parent axis*)}. The second plan is created from the first plan by permutating the steps 2 and 4. The total times for loading in these plans are 3109ms and 2844ms, respectively. The times for querying in both plans are equal. Therefore, the second plan is better than the first one. The result accords with our rule in Section 4.2 for query plan selection based on the number of connections between processing modules and RDBMS.

- When the keywords are issued, the sizes of buffers decrease so the joint costs in main memory of SKEYRUS becomes small.

- For the queries including the axes *preceding* or $following$, such as Q4, the evaluation cost was high. The runs for the query were cancelled in Xalan.

6 Conclusion

Queries on keywords are common user requests whereas XML makes the queries on document structure available. Therefore, it is natural to integrate these tasks to enrich the selectivity of users. In this study, we described SKEYRUS, a system which enables the searches on both keywords and structure of XML documents. The input of SKEYRUS is XPath expressions having word-containment predicates, therefore the query expressiveness is significantly extended. The core technique of SKEYRUS is the rUID numbering scheme, which has been designed to be scalable and robust in structural update. The ability of rUID in XPath axes construction has been exploited in query processing. The preliminary experiments have shown the efficiency of rUID and SKEYRUS. The query processing in main memory of SKEYRUS is better than of Xalan.

Extensions of this study are underway, including the implementation of an indexing engine based on the B^+-tree structure to index the data tables in SKEYRUS in order to avoid the limitations in using an underlying RDBMS. We also plan to conduct more performance tests in different configurations.

References

[1] S.Chien, et al. Storing and Querying Multiversion XML Documents using Durable Node Numbers. Proc. of the Inter. Conf. on WISE, Japan, 270–279, 2001.

[2] P.F.Dietz. Maintaining order in a link list. Proceeding of the Fourteenth ACM Symposium on Theory of Computing, California, 122–127, 1982.

[3] D.D.Kha, M.Yoshikawa, S.Uemura: A Structural Numbering Scheme for XML Data. EDBT workshop on XML Data Management (XMLDM), Prague, March 2002.

[4] D.Florescu, I. Manolescu, D.Kossmann. Intergrating Keyword Search into XML Query Processing. Proc. of the Inter. WWW Conf., Elsevier ,Amsterdam, 2000.

[5] A.Marian, S.Abiteboul, G.Cobena, L.Mignet. Change-Centric Management of Versions in an XML Warehouse. Proc. of the Inter. Conf. on VLDB, Italy, 2001.

[6] H.Jang, Y.Kim, D.Shin. An Effective Mechanism for Index Update in Structured Documents. Proc. of CIKM, USA, 383–390, 1999.

[7] D.Shin. XML Indexing and Retrieval with a Hybrid Storage Model. J. of Knowledge and Information Systems, 3:252–261, 2001.

[8] J.McHugh, J.Widom. Query Optimization for XML. Proc. of the Inter. Conf. on VLDB, Edinburgh, Scotland, pages 315–326, 1999.

[9] J.Shanmugasundaram, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. Proc. of the Inter Conf. on VLDB, Scotland, 1999.

[10] Y.K.Lee, S-J.Yoo, K.Yoon, P.B.Berra. Index Structures for structured documents. ACM First Inter. Conf. on Digital Libraries, Maryland, 91–99, 1996.

[11] Q.Li, B.Moon. Indexing and Querying XML Data for Regular Path Expressions. Proc. of the Inter. Conf. on VLDB, Italy, 2001.

[12] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml, 2000.

[13] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 2000.

[14] Apache Software Foundation. Apache XML Project. http://xml.apache.org/, 2001.

[15] M.Yoshikawa, T.Amagasa, T.Shimura, S.Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM Transation on Internet Technologies, 1(1), 2001.

[16] C.Zhang, et al. On Supporting Containment Queries in Relational Database Management Systems. Proc. of the ACM SIGMOD, USA, 2001.