The SQL Function XMLNEST to Generate Recursive XML Values from Relational Data

SUNG-CHUL PARK AND YOUNG-CHUL PARK
School of Computer
Kyungpook National University
Daegu, 702-701 Korea

SQL standard provides SQL/XML publishing functions to publish the result of an SQL query as XML values but it does not provide any SQL/XML publishing function that can publish the result of a recursive SQL query as recursively structured XML values. Therefore, to publish transitively connected relational tuples as recursively structured XML values with the use of appropriate SQL/XML publishing functions, we have to write a nested SQL query. Writing that query, however, is not easy provided that the depth of the connections is deep even if we know the depth of them and is not possible once the depth of the connections is not known in advance. To solve the problem, we propose a new SQL function XMLNEST that can publish the result of a recursive SQL query as recursively structured XML values. Both the recursive structure and the order of sibling XML elements of the XML values can be specified in the invocation of XMLNEST. Our experiments show that the proposed scheme will be the unique reasonable solution to the problem.

Keywords: XML schema graph, XQuery, SQL/XML publishing function, recursive query, nested query, recursive XML values, SQL aggregate function, XMLNEST

1. INTRODUCTION

There have been a number of studies on publishing relational data in the database as XML values [1, 2, 3, 4]. SQL/XML, which is XML related specifications of SQL standard [5], defines the XML data type “XML” as a native SQL data type and SQL/XML publishing functions [6] such as XMLELEMENT, XMLATTRIBUTES, XMLAGG, XMLFOREST, and XMLCONCAT as SQL functions to publish the result of an SQL query as XML values. SQL standard, however, does not support any SQL/XML publishing function that can be used in the recursive SQL query [7, 8, 9] to publish the result of the query as recursively structured XML values, which we call recursive XML values. Therefore, for some specific relational tuples and the relational
tuples that are connected with the specific tuples either directly or indirectly according to
given conditions, to publish them as recursive XML values with the use of SQL/XML
publishing functions, we have to write a nested SQL query. DB2 [10, 11] and Oracle
[12], to name just a few among commercial database management systems, support
SQL/XML of SQL standard.

Table 1 illustrates the structures and tuples of tables EMPLOYEE and
DEPARTMENT, which are referenced by the queries in this paper. Note that
employee_id (department_id, respectively) is the primary key of table EMPLOYEE
(DEPARTMENT, respectively).

Table 1. The structures and tuples of tables EMPLOYEE and DEPARTMENT.

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th></th>
<th></th>
<th></th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee-id</td>
<td>name</td>
<td>salary</td>
<td>supervisor-id</td>
<td>manager-id</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>35000</td>
<td>NULL</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>30000</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>40000</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>45000</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>20000</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>25000</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>G</td>
<td>50000</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>K</td>
<td>25000</td>
<td>NULL</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>L</td>
<td>20000</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>M</td>
<td>27000</td>
<td>21</td>
<td>4</td>
</tr>
</tbody>
</table>

**Example 1.** In table EMPLOYEE, for the employees of employee_id 10 and 21 and their
subordinates up to and including two-level behind the two employees (i.e., subordinates
of the employees and subordinates of the subordinates of the employees), publish the
information of them and the relationship among them as XML values.

A nested SQL query using SQL/XML publishing functions for Example 1 is shown
in Fig. 1.
Fig. 1. A nested SQL query using SQL/XML publishing functions.

The two result tuples of the query of Fig. 1 are shown in Fig. 2, where XML element “EMPLOYEE” has an XML attribute “id” and subelements “NAME,” “SALARY,” “MANAGER,” “DEPT,” and “SUBORDINATES.”

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;EMPLOYEE id=&quot;10&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;A&lt;/NAME&gt;&lt;SALARY&gt;35000&lt;/SALARY&gt;&lt;DEPT&gt;HR&lt;/DEPT&gt;</td>
</tr>
<tr>
<td>&lt;SUBORDINATES&gt;</td>
</tr>
<tr>
<td>&lt;EMPLOYEE id=&quot;11&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;B&lt;/NAME&gt;&lt;SALARY&gt;30000&lt;/SALARY&gt;&lt;MANAGER&gt;10&lt;/MANAGER&gt;&lt;DEPT&gt;HR&lt;/DEPT&gt;</td>
</tr>
<tr>
<td>&lt;/EMPLOYEE&gt;</td>
</tr>
<tr>
<td>&lt;EMPLOYEE id=&quot;12&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;C&lt;/NAME&gt;&lt;SALARY&gt;40000&lt;/SALARY&gt;&lt;MANAGER&gt;10&lt;/MANAGER&gt;&lt;DEPT&gt;RD&lt;/DEPT&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&lt;/SUBORDINATES&gt;</td>
</tr>
<tr>
<td>&lt;/EMPLOYEE&gt;</td>
</tr>
<tr>
<td>&lt;EMPLOYEE id=&quot;21&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;K&lt;/NAME&gt;&lt;SALARY&gt;25000&lt;/SALARY&gt;&lt;DEPT&gt;TFT&lt;/DEPT&gt;</td>
</tr>
<tr>
<td>&lt;SUBORDINATES&gt;</td>
</tr>
<tr>
<td>&lt;EMPLOYEE id=&quot;22&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;L&lt;/NAME&gt;&lt;SALARY&gt;20000&lt;/SALARY&gt;&lt;MANAGER&gt;21&lt;/MANAGER&gt;&lt;DEPT&gt;TFT&lt;/DEPT&gt;</td>
</tr>
<tr>
<td>&lt;/EMPLOYEE&gt;</td>
</tr>
<tr>
<td>&lt;EMPLOYEE id=&quot;23&quot;&gt;</td>
</tr>
<tr>
<td>&lt;NAME&gt;M&lt;/NAME&gt;&lt;SALARY&gt;27000&lt;/SALARY&gt;&lt;MANAGER&gt;21&lt;/MANAGER&gt;&lt;DEPT&gt;TFT&lt;/DEPT&gt;</td>
</tr>
<tr>
<td>&lt;/EMPLOYEE&gt;</td>
</tr>
<tr>
<td>&lt;/SUBORDINATES&gt;</td>
</tr>
<tr>
<td>&lt;/EMPLOYEE&gt;</td>
</tr>
</tbody>
</table>

Fig. 2. The two result tuples of the query of Fig. 1.

The structure of XML values in Fig. 2 can be illustrated as an XML schema graph, which is shown in Fig. 3.
 Provided that an XML schema graph illustrating the structure of XML values formulates a cycle of nodes and edges, we define that the graph represents recursive XML values. In an XML schema graph having a cycle, we represent the start node of the cycle as a double ellipse. In Fig. 3, since nodes “EMPLOYEE” and “SUBORDINATES” and edges connecting the two nodes formulate a cycle, we say that XML values in Fig. 2 are recursive XML values.

Since Example 1 restricts the scope of subordinates to two-level behind the start employees, as far as that request is concerned, it is simple to write a nested SQL query. However, suppose we want either to restrict the scope of someone’s subordinates to ten-level or to put no boundary condition on that scope. Writing a nested SQL query like Fig. 1 for the former is not easy and doing that for the latter is impossible. The contributions of this paper are as follows.

First, we classify groups of tuples that are identified by an SQL query into two kinds: one is the traditional concept of the group and the other is the hierarchy group. The hierarchy group is our new proposal for the group, which is a set of tuples that consists of both a specific tuple and the tuples that are connected with the specific tuple either directly or indirectly according to given conditions.

Second, to publish result tuples of a recursive SQL query as recursive XML values, we propose a new SQL function $\text{XMLNEST}$ that can be used in the query. Once $\text{XMLNEST}$ is used in a recursive SQL query, tuples identified by the query are grouped according to the notion of the hierarchy group and for each hierarchy group one recursive XML value is built.

Third, to get aggregate values against subsets of the tuples in a hierarchy group, we propose five new SQL functions, which we call $\text{XMLNEST-related functions}$. Those functions can be used as direct and indirect arguments of $\text{XMLNEST}$.
The rest of this paper is organized as follows. Section 2 describes and analyzes two feasible methods that produce recursive XML values without using SQL/XML publishing functions. In Section 3, some characteristics of the recursive SQL query are summarized; some important concepts for building recursive XML values from the recursive SQL query are presented; and XMLNEST and XMLNEST-related functions are proposed. In Section 4, the performance of the recursive SQL queries using XMLNEST is evaluated and compared with other schemes. Section 5 summarizes and concludes this paper.

2. OTHER METHODS FOR PRODUCING RECURSIVE XML VALUES

The methods that produce recursive XML values without using SQL/XML publishing functions can be classified into two types. The first type is making stored procedures, stored functions, or database application programs that do the following works: (1) formulate a recursive SQL query, (2) declare a cursor on the query, (3) for each tuple that is fetched through the cursor, generate XML elements by attaching tags to the column values of the tuple, and (4) arrange the generated XML elements according to their connection relationship. This method has the following characteristics. First, for each result tuple of the query, result tuples that are connected with the result tuple either directly or indirectly have to be identified. This means that application programmers have to develop codes for the task that is similar to generating result tuples of the recursive SQL query. Since that identification of connected result tuples can be done based on column values of the result tuples, some additional columns just for that identification may need to be included in the query itself. Second, the order of result tuples of the query may or may not be identical with the document order [13] of the recursive XML values to be created. Once they are not identical, application programmers have to sort the result tuples. Third, in the XML schema graph that illustrates the structure of recursive XML values, if the child node that formulates a cycle with a parent node is not the last child node of the parent node, the application program becomes much more complicate.

The second type for producing recursive XML values is utilizing the XQuery query [14]. Since the database language SQL is based on the relational model and XQuery is
based on the XQuery/XPath data model [15], SQL and XQuery cannot be integrated into one language. To solve that problem, SQL standard [5] defines XQuery functions such as XMLQUERY, XMLTABLE, and XMLEXISTS as SQL functions. Those XQuery functions support the XQuery query against XML values stored in the database such that they produce XML values, SQL values, and Boolean values respectively as their results. To generate recursive XML values from relational tuples by utilizing XQuery queries, either the XQuery query using nested FLWOR (for, let, where, order by, and return) expressions or the XQuery query using recursive user defined functions against implicit XML views [1, 16] on relational tuples can be used.

It is simple to write an XQuery query using nested FLWOR expressions that produces the same information as the nested SQL query using SQL/XML publishing functions of Fig. 1 does. However, as noted before at the discussion of the nested SQL query, writing an XQuery query using nested FLWOR expressions for the tuples having known but very deep connection depth is not easy and doing that for the tuples of unknown connection depth is impossible. For the XQuery query using recursive user defined functions, developers have to code recursive user defined functions that reflect contents and structures of the XML values to be created and then write an XQuery expression that contains invocations of the user defined functions. The recursive user defined functions, for example the recursive XQuery query of Oracle, should have the following characteristics: (1) they put implicit XML views on relational tables to be queried, (2) by querying the views through XQuery expressions, they identify XML elements that correspond to some specific relational tuples and the relational tuples that are connected with the specific tuples either directly or indirectly, (3) they produce new XML elements of the intended XML value structure based on the identified XML elements, and (4) they invoke themselves in that XQuery expressions. The user defined functions have to be coded as recursive functions since it is not possible to predict the connection depth among XML elements. Depending on the values to be built, the user defined functions can be formulated as FLWOR expressions, conditional expressions, arithmetic expressions, and others.

The method of using the XQuery query for publishing relational data as recursive XML values may not be a good solution because of its execution time. Specifically, the XQuery query using nested FLWOR expressions cannot be a good choice and the
XQuery query using recursive user defined functions should not be taken as a solution for that purpose. The performance materials related to the XQuery queries are described in Section 4 of this paper.

3. XMLNEST

This section summarizes some characteristics of the recursive SQL queries and then defines three important concepts, which we call RT-trees, RRT-trees, and hierarchy groups of tuples that are tightly related to the result tuples of recursive SQL queries. After that, based on the concepts, XMLNEST and XMLNEST-related functions are presented. The SQL function XMLNEST produces recursive XML values for each of set of tuples of the transitive closures of a recursive SQL query.

3.1 Review of Recursive SQL Queries

Recursive SQL queries in database management systems are select statements and are performed based on the transitive closure. In a recursive SQL query, we define the tuples that are identified by the transitive closure declared at the query as recursive tuples (RTs) and the result tuples of the query as recursive result tuples (RRTs). The set of RTs of a recursive SQL query consists of the tuples that represent the start of the transitive closure of the query, which we call start tuples, and the tuples that are connected to the start tuples either directly or indirectly by the transitive closure of the query. In this paper, for three arbitrary RTs, say $t_a$, $t_b$, and $t_c$, if tuples $t_b$ and $t_c$ are identified by the transitive closure directly from $t_a$, we say that $t_a$ is called the parent tuple of tuples $t_b$ and $t_c$; tuples $t_b$ and $t_c$ are called child tuples of $t_a$; the relationship between $t_a$ and tuples $t_b$ and $t_c$ is called the parent-child relationship; and tuples $t_b$ and $t_c$ are called sibling tuples. In addition to that, for two arbitrary RTs, say $t_a$ and $t_b$, if $t_b$ is identified by the transitive closure either directly or indirectly from $t_a$, we say that $t_a$ is called an ancestor tuple of $t_b$; $t_b$ is called a descendant tuple of $t_a$; and the relationship between tuples $t_a$ and $t_b$ is called the ancestor-descendant relationship. Among commercial database management systems, DB2 [17, 18] and SQL Server [19, 20], to name just a few, support recursive SQL queries of SQL standard; and Oracle supports them with Oracle’s own syntax [16].
The transitive closure can be specified in recursive SQL queries of SQL standard as follows.

1. To identify the start tuples of the transitive closure, both the tables from which the start tuples can be identified, which we call foundation tables for the start tuples, and the conditions (if any) that are to be applied to the foundation tables to identify the start tuples, which we call root_conditions, have to be declared.

2. To identify all the tuples that are connected to the start tuples either directly or indirectly from the start tuples, both the tables, which we call foundation tables for finding descendant tuples of the start tuples, and the conditions that are to be applied to the foundation tables and also to the start tuples to identify descendant tuples of the start tuples, which we call parent_child_conditions, have to be declared.

In the recursive SQL queries of the SQL standard, through the with clause, the transitive closure and some temporary tables to keep the RTs that are identified by the transitive closure can be specified, and through the closing subquery of the query, for the production of RRTs, in addition to the temporary tables that contain the found RTs, some additional tables, which we call reference tables, and some conditions, which we call RT_reference_conditions, that contain conditions that the found RTs in the temporary tables and the tuples in the reference tables have to satisfy can be specified.

The recursive SQL query of Oracle is a select statement that can have a start-with clause to specify the root_conditions, a connect-by clause to specify the parent_child_conditions, and order-siblings-by clause to specify the order of sibling tuples in the result set of the query. The foundation tables that are used to identify both the start tuples and the descendant tuples of the start tuples are specified in the from clause of the query; and RT_reference_conditions and part of both the root_conditions and the parent_child_conditions that can be used just for join conditions among foundation tables can be specified in the where clause of the query.

For the illustration of the recursive SQL query of SQL standard and that of Oracle, let’s consider Example 2.

**Example 2.** From tables EMPLOYEE and DEPARTMENT of Table 1, for each employee of employee_id 10 and 21, find him/her and his/her subordinates such that retrieve columns <employee_id, name, salary, supervisor_id, department_name> of the
tuples whose salary is less than 40000.

(a) A recursive SQL query of SQL standard. (b) A recursive SQL query of Oracle.

Fig. 4. Recursive SQL queries.

The request of Example 2, like that of Example 1, wants to identify employees on the transitive closure. However different from Example 1, (1) Example 2 does not restrict the depth of the subordinates such that all the employees on the transitive closure should be identified, and (2) Example 2 has some restriction on the employees on the transitive closure such that some of the employees may not be put on the result. The query of Fig. 4 (a), which is an example of the recursive SQL query of SQL standard, identifies and keeps RTs of the query into a named temporary table. However, the query of Fig. 4 (b), which is an example of the recursive SQL query of Oracle, identifies and keeps RTs of the query into an implicitly defined temporary table.

3.2 RT-trees, RRT-trees, and Hierarchy Groups of Tuples

To simplify our discussion of the concepts that are to be presented in this section, without loss of generality, we assume that there is not any cyclic connection among RTs. In other words, once an RT, say \( t_a \), becomes an ancestor of another RT, say \( t_b \), we assume that \( t_a \) cannot be a descendant of \( t_b \).

**Definition 1 (RT-tree).** For the RTs of a recursive SQL query, (1) by representing the RTs as nodes and, (2) for each pair of nodes corresponding to each pair of RTs of parent-child relationship among the RTs, by putting a directed edge from the node of the

---

1 In the recursive SQL query of Oracle, the NOCYCLE option in the connect-by clause signifies that once a cyclic connection is found, that connection is just ignored. However, without the specification of the NOCYCLE option, once a cyclic connection is found, the query itself is rejected. SQL Server confines the cyclic connection to the value of maxrecursion. However, DB2 does not put any restriction on the cyclic connection.
parent RT to the node of the child RT and (3) if the sort order among sibling tuples is specified, by putting the nodes corresponding to the sibling RTs from left to right according to the order and otherwise, by putting the nodes of the sibling RTs in an arbitrary order, trees are built. We call those trees recursive tuples trees (RT-trees) and name RTs that correspond to the root nodes of RT-trees root RTs of the trees.

For example, for the query of Fig. 4 (b), the set of start tuples, saying $\Delta_0$, becomes $\sigma_{\text{employee_id}=10 \text{ or } \text{employee_id}=21}(\text{EMPLOYEE} \bowtie_{\text{department_id}=\text{department_id}} \text{DEPARTMENT}) = \{<10, A, 35000, \text{NULL}, \text{HR}>, <21, K, 25000, \text{NULL}, \text{TFT}>\}$ and the set of connected tuples is defined to be $\Delta_{i+1} = (\Delta_i \bowtie_{\text{employee_id}=\text{supervisor_id}} (\text{EMPLOYEE} \bowtie_{\text{department_id}=\text{department_id}} \text{DEPARTMENT}))$ such that the set of direct subordinates of the start tuples, saying $\Delta_1$, becomes $\{<11, B, 30000, 10, \text{HR}>, <12, C, 40000, 10, \text{RD}>, <22, L, 20000, 21, \text{TFT}>, <23, M, 27000, 21, \text{TFT}>\}$, the set of subordinates who are two-level behind the start tuples, saying $\Delta_2$, becomes $\{<13, D, 45000, 12, \text{RD}>, <14, E, 20000, 12, \text{SALES}>, <15, F, 25000, 13, \text{HR}>, <16, G, 50000, 13, \text{RD}>\}$, and $\Delta_4 = \{\}$. Therefore, the final set of RTs of the query becomes $\Delta_0 \cup \Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta_4$ and it can be visualized as two RT-trees as shown in Fig. 5. In the figure, to identify nodes conveniently, each node is labelled with the value of column name of the node’s corresponding RT.

**Fig. 5. RT-trees for the query of Fig. 4 (b).**

**Definition 2 (Original RTs and reincarnated RRTs).** In a recursive SQL query, suppose that an RRT, say $t_a'$, has been produced by an RT, say $t_a$. Then $t_a$ is called an original RT of $t_a'$ and $t_a'$ is a reincarnated RRT of $t_a$.

\footnote{Even though the order-siblings-by clause of the recursive SQL query of Oracle, for example, specifies the sort order among sibling RRTs, the sort order can also be applied to the sibling RTs in advance without violating the meaning.}
Definition 3 (Participating RTs and non-participating RTs). Among RTs of a recursive SQL query, the RTs that become original RTs of some RRTs are called participating RTs, and the RTs that do not have any reincarnated RRT of them are called non-participating RTs.

For example, among RTs of Fig. 5, nodes C, D, and G do not satisfy the predicate “e.salary < 40000” of the query of Fig. 4 (b) such that they become non-participating RTs. All the other RTs of Fig. 5 are participating RTs.

Depending on the conditions that are used for the production of RRTs, an RRT can have several original RTs of it. Therefore, depending on the number of RTs that have participated in the production of an RRT, we classify the production into two kinds: the replication of RTs and the bonds among RTs.

Definition 4 (RRT production by the replication of RTs). If an RRT of a recursive SQL query is produced by exactly one temporary table that is defined by the query and some reference tables (if any), we say that the RT of the temporary table.

Definition 5 (RRT production by bonds among RTs). If an RRT of a recursive SQL query is produced by more than two temporary tables that are defined by the query and some reference tables (if any), we say that the RRT is produced by bonds among RTs of the temporary tables.

The replication of RTs can be classified further into single-replication and multiple-replication depending on the number of reincarnations of the original RTs. If an RT is replicated into exactly one RRT, that replication is called a single-replication; and if it is replicated into more than two RRTs, that replication is called a multiple-replication. For example, the closing subquery in the query of Fig. 4 (a) has only one recursive temporary table empHierarchy, one reference table DEPARTMENT, and the RT_reference_conditions of “e.salary < 40000 and e.department_id = d.department_id”, where the column department_id is the primary key of table DEPARTMENT and the column department_id of table EMPLOYEE is a foreign key that references the column department_id of table DEPARTMENT such that each RT produces at most one RRT. Therefore, RRTs of the query are produced by the single-replication of RTs.
Like RTs, RRTs of recursive SQL queries can have their relative position among them. For that purpose, we define the parent-child relationship and the ancestor-descendant relationship among RRTs in definitions 6 and 7.

**Definition 6 (Parent-child relationship among RRTs).** For two arbitrary RTs, say $t_p$ and $t_c$, that are in the parent-child relationship, assuming that $t_p$ is the parent tuple of $t_c$, if one of the reincarnated RRTs of $t_p$ is $t_p'$ and that of $t_c$ is $t_c'$, we say that $t_p'$ and $t_c'$ are in the parent-child relationship, $t_p'$ is the parent tuple of $t_c'$, and $t_c'$ is a child tuple of $t_p'$.

**Definition 7 (Ancestor-descendant relationship among RRTs).** For two arbitrary RTs, say $t_a$ and $t_d$, that are in the ancestor-descendant relationship, assuming that $t_a$ is an ancestor tuple of $t_d$, if one of the reincarnated RRTs of $t_a$ is $t_a'$ and that of $t_d$ is $t_d'$, we say that $t_a'$ and $t_d'$ are in the ancestor-descendant relationship, $t_a'$ is an ancestor tuple of $t_d'$, and $t_d'$ is a descendant tuple of $t_a'$.

**Definition 8 (RRT-graph).** For the RRTs of a recursive SQL query, (1) by representing the tuples as nodes and, (2) for each pair of nodes corresponding to each pair of RRTs of the parent-child relationship among the RRTs, by putting a directed edge from the node of the parent RRT to the node of the child RRT and (3) if the sort order among sibling tuples is specified, by putting the nodes corresponding to the sibling RRTs from left to right according to the order and otherwise, by putting the nodes of the sibling RRTs in an arbitrary order, graphs are built. We call those graphs recursive result tuples graphs (RRT-graphs).

The purpose of the SQL function XNLNEST lies in publishing RRTs as XML values such that we need to keep the RRT-graphs as trees, which we call RRT-trees, and formulate at most one RRT-tree from RRTs generated by the RTs of an RT-tree. For that purpose, we put the following three restrictions to the production of RRTs.

**Restriction_1.** We restrict the notion of the parent-child relationship and the ancestor-descendant relationship among RRTs into those relationships among RRTs that are produced by single-replications of RTs.

The reasons of having Restriction_1 are as follows.

(1) Taking join operations between two sets of RTs of the transitive closures does not imply any semantic significance. We just want to build recursive XML values for
each set of RTs of the transitive closures while supplementing each RT with some information. Therefore, we reject the RRT production by bonds among RTs.

(2) The RRT production by the multiple-replication of RTs might not keep the RRT-graphs as trees. For example, for two RTs, saying \( t_p \) and \( t_c \), that are in the parent-child relationship, assuming that \( t_p \) is the parent tuple of \( t_c \), if \( t_p \) has two reincarnated RRTs \( t_p' \) and \( t_p'' \), and \( t_c \) has only one reincarnated RRT \( t_c' \), then \( t_c' \) has two parents such that the RRT-graph of \( t_p', t_p'', \) and \( t_c' \) cannot be a tree. Once the RRT-graphs are not kept as trees, we might lose the tree characteristic of the transitive closure such that transitively connected relational tuples cannot be produced as recursive XML values.

According to Restriction_1, once RRTs are produced either by multiple-replications of RTs or by bonds among RTs, applying XMLNEST to the RRTs causes the query to be rejected. Therefore, once the query satisfies Restriction_1, each RT of any set of RTs of the transitive closures of the query produces at most one reincarnated RRT of it and each RRT that is produced by any RT of the set is produced by exactly one RT. \(^3\)

**Restriction_2.** For each RT of an RT-tree, if it is a non-participating RT and does not have any participating RT as its descendant, no RRT is produced for it.

**Restriction_3.** For each RT of an RT-tree, if it is a non-participating RT and has at least one participating RT as its descendant, we produce a NULL-valued RRT for it.

NULL-valued RRTs are RRTs, have the same fields as normal RRTs, and have NULL values for all the database columns. The NULL-valued RRT for a specific RT, say \( t_x \), is a child RRT of the RRT that is produced for the parent RT (if any) of \( t_x \), and is the parent RRT of the RRTs that are produced for the child RTs (if any) of \( t_x \).

Therefore, once a query that satisfies Restrictions 1, 2 and 3 is given, at most one RRT-tree can be produced for each RT-tree of the query and the RRT-trees keep the topological relationship among non-NUL L-valued RRTs. Fig. 6 illustrates two RRT-trees that are formulated by the query of Fig. 4 (b). In the two RRT-trees of Fig. 6, which are derived from the two RT-trees of Fig. 5, (1) no RRT is produced for RT \( G \)
since RT $G$ is a non-participating RT and does not have any participating RT as its descendant, (2) a NULL-valued RRT is produced for RT $D$ since RT $D$ is a non-participating RT and RT $F$ that is a descendant RT of RT $D$ is a participating RT, and (3) another NULL-valued RRT is produced for RT $C$ since RT $C$ is a non-participating RT and RTs of $E$ and $F$ are participating RTs.

![Diagram of RRT-trees for the query of Fig. 4 (b).](image)

**Fig. 6.** RRT-trees for the query of Fig. 4 (b).

**Definition 9 (Hierarchy group of tuples).** For an RRT-tree of a recursive SQL query, among all the RRTs of the RRT-tree, let the set of non-NULL-valued RRTs be $S$. Provided that $S$ is not empty, we call the RRTs in $S$ a hierarchy group of tuples.

In Fig. 6, for the RRT-tree rooted at RRT $A$, the set of RRTs \{A, B, E, F\} formulates a hierarchy group, and for the RRT-tree rooted at RRT $K$, the set of RRTs \{K, L, M\} formulates another hierarchy group. By the introduction of the hierarchy group, groups of tuples that are formulated by a query can be classified into two kinds: one is the traditional concept of the group and the other is the hierarchy group. For this classification, we call tuples of the traditional concept of the group, which are built either by specifying grouping columns on the *group-by* clause of the select statement or by specifying any of the SQL aggregate functions of SUM, COUNT, AVG, MIN, MAX, and XMLAGG but not having the group-by clause, the *same-value group* of tuples.

### 3.3 XMLNEST and XMLNEST-related Functions

\[\text{All the RRTs that are produced by the recursive SQL query of Oracle are produced by single-replication of RTs. That is because exactly one temporary table for keeping RTs is implicitly defined and no reference table can be specified at the query.}\]
The SQL function XMLNEST can be used at the select clause of the recursive SQL queries and is devised to generate XML values for RRT-trees that are formulated by the queries. The specification of XMLNEST has a tight relationship with XML schema graphs that represent recursive XML values. In an XML schema graph having one cycle, among the nodes participating in the cycle, the start node of the cycle is called NEST_ROOT node, and the node that constitutes the cycle with the NEST_ROOT node is called NEST_SUBORDINATE node. Note that the NEST_SUBORDINATE node is a child node of the NEST_ROOT node and does not have any other child node except for the NEST_ROOT node. For example, in the XML schema graph in Fig. 3, node “EMPLOYEE” is the NEST_ROOT node and node “SUBORDINATES” is the NEST_SUBORDINATE node. In XML values that are represented by an XML schema graph having one cycle, XML elements that the NEST_ROOT node of the graph represents are called NEST_ROOT elements and XML elements that the NEST_SUBORDINATE node of the graph represents are called NEST_SUBORDINATE elements. Fig. 7 shows the syntax of XMLNEST.

```
<XML nest> ::=  
  XMLNEST <left paren> 
  <XML nest_root element spec> 
  [[<comma> <XML element content>]…] 
  <comma> <XML nest_subordinate element spec> 
  [[<comma> <XML element content>]…] <right paren> 

<XML nest_root element spec> ::=  
  [NAME] <XML element name> 
  [[<comma> <XML namespace declaration>] 
  [<comma> <XML attributes>]] 

<XML nest_subordinate element spec> ::=  
  XMLNESTSPEC <left paren> 
  [[NAME] <XML element name> [OPTIONAL | MANDATORY] 
  [<comma> <XML attributes>]] 
  [ORDER BY <sort specification list> [NULLS FIRST | NULLS LAST] ] <right paren> 
```

Fig. 7. Specification of XMLNEST.

In Fig. 7, <XML nest_root element spec> specifies the name of the NEST_ROOT element, XML namespaces of the element, and XML attributes of the element; and <XML element content> specifies each child of the NEST_ROOT element except for the
NEST_SUBORDINATE element. The meanings of the required syntax for `<XML
nest_subordinate element spec>` in Fig. 7 are as follows.

1. `<XML element name>` means the name of the NEST_SUBORDINATE element. That name can be omitted. Once it is specified, for each non-terminal node of RRT-trees, a NEST_SUBORDINATE element is produced and that element becomes the parent element of NEST_ROOT elements that are produced for all child nodes of the non-terminal node. Once it is omitted, NEST_SUBORDINATE elements are not produced and for each non-terminal node of RRT-trees, NEST_ROOT elements for all the child nodes of the non-terminal node are produced as subelements of the NEST_ROOT element for the non-terminal node.

2. The illustration of the NEST_SUBORDINATE element can be either “OPTIONAL” or “MANDATORY.” The default is “MANDATORY.” “MANDATORY” means that every NEST_ROOT element has exactly one NEST_SUBORDINATE element; and “OPTIONAL” means that NEST_ROOT elements for non-terminal nodes of RRT-trees have NEST_SUBORDINATE elements but those for terminal nodes do not have NEST_SUBORDINATE elements.

3. `<XML attributes>` specifies XML attributes of the NEST_SUBORDINATE element.

4. The ORDER BY keyword followed by the sort specification list specifies the sort criterion of sibling NEST_ROOT elements. Once it is specified, for each non-terminal node of RRT-trees, NEST_ROOT elements for all child nodes of the non-terminal node are ordered according to the sort criterion. Otherwise, the NEST_ROOT elements are listed by the order of their corresponding child RRT nodes in the RRT-tree from left to right. For ordering sibling NEST_ROOT elements having NULL values for the columns in the sort specification list, NULLS FIRST or NULLS LAST can be used.

In Fig. 7, the required syntaxes for the specifications of `<XML element name>`,
`<XML namespace declaration>`, `<XML attributes>`, `<XML element content>`, and `<sort specification list>` are the same as those of SQL/XML. Note that Fig. 7 does not contain the functionality of limiting the depth of RRT-trees. That functionality can be supported by the recursive SQL queries themselves.
SQL supports aggregate functions of SUM, COUNT, AVG, MIN, MAX, and XMLAGG that have been applied to the same-value groups of tuples. Since XMLNEST works for the hierarchy groups of tuples, the existing SQL aggregate functions and XMLNEST should not be used together at the same subquery. However, for the acquisition of diverse aggregate values for the subtrees of RRT-trees, we propose five more hierarchical aggregate functions of NEST_SUM, NEST_COUNT, NEST_AVG, NEST_MIN, and NEST_MAX that can be used as direct and indirect arguments of XMLNEST. We call those five hierarchical aggregate functions *XMLNEST-related functions*. The specification and meaning of each of the XMLNEST-related functions are the same as those of the existing SQL aggregate function that corresponds to the name of the XMLNEST-related function by dropping the prefix “NEST_” from that name. Once the XMLNEST-related functions are used as arguments of XMLNEST, upon treating an RRT that corresponds to a node of an RRT-tree, whether it is a NULL-valued RRT or not, XMLNEST-related functions calculate value expressions specified as their arguments against the set of the non-NULL-valued RRTs in the subtree of the RRT-tree rooted at the node and make the node have that value.

The effects of XMLNEST on the existing SQL/XML publishing functions can be summarized as follows. First, only the SQL/XML publishing functions that can produce XML values for each RRT can be used as direct and indirect arguments of XMLNEST. Among the SQL/XML publishing functions, XMLELEMENT, XMLCONCAT, XMLFOREST, and XMLATTRIBUTES satisfy that property such that they can be used as arguments of XMLNEST. Second, since XMLNEST returns XML values against RRT-trees, XMLNEST can be used as arguments of the SQL/XML publishing functions that can have XML value expressions as their arguments. Among the SQL/XML publishing functions, XMLELEMENT, XMLCONCAT, and XMLFOREST satisfy that property such that they can accommodate XMLNEST as their arguments.

**Example 3.** Based on the recursive SQL query of Example 2, make XML values that reflect the information of each employee that is found by the query including the total salary and the member count for the employee and his/her subordinates.

Fig. 8 shows a recursive SQL query based on the syntax of Oracle for Example 3.
Fig. 8. A recursive SQL query with XMLNEST and XMLNEST-related functions.

RT-trees and RRT-trees that the query of Fig. 8 generates are the same as those in Fig. 5 and Fig. 6 respectively. By applying XMLNEST to each RRT-tree in Fig. 6, two final result tuples in Fig. 9 are produced.

```
SELECT XMLNEST("EMPLOYEE", XMATTRIBUTES(employ_id AS "id")),
       XMNELEMT("NAME", name),
       XMNELEMT("SALARY", salary),
       XMNELEMT("MANAGER", supervisor_id),
       XMNELEMT("DEPT", department_name),
       XMNELEMT("SUM_SALARY", NEST_SUM(salary)),
       XMNELEMT("COUNT", NEST_COUNT(1)),
       XMNELEMT("SUBORDINATES", "ORDER BY employ_id")
FROM employee e, department d
WHERE e.salary < 40000 AND e.department_id = d.department_id
START WITH e.employ_id = 10 OR e.employ_id = 21
CONNECT BY PRIOR e.employ_id = e.supervisor_id
```

Fig. 9. The final result tuples of the query of Fig. 8.

Note that even though NULL-valued RRTs have NULL values for their database column values, they possess the returned values from the XMLNEST-related functions.
For example, in the XML values in Fig. 9, the XML element “EMPLOYEE” for node C in Fig. 6 has NULL values for its attribute “id” and subelements of “NAME,” “SALARY,” “MANAGER,” and “DEPT.” However, it has values of 45000 and 2 for its subelements of “SUM_SALARY” and “COUNT,” respectively.

4. PERFORMANCE EVALUATIONS

To evaluate the functionality and performance of XMLNEST, we have implemented the recursive SQL query of Oracle, SQL/XML publishing functions, XMLNEST, and XMLNEST-related functions on SQLite [21]. In this section, for the RRT-trees of some known depth, four different types of queries such as (1) XQuery queries using nested FLWOR expressions, (2) XQuery queries using recursive user defined functions, (3) nested SQL queries using SQL/XML publishing functions, and (4) recursive SQL queries using XMLNEST are evaluated. The two types of the XQuery queries and the nested SQL queries are evaluated on Oracle 10g, and the nested SQL queries and the recursive SQL queries are evaluated on SQLite.

Our experiments have been performed on a standard personal computer with an Intel Core 2 Duo E7500 2.93GHz processor, 1.96GB of main memory, 300GB of hard disk and the Microsoft Windows XP Professional Version 2002 Service Pack 3 operating system. We have used tables of EMPLOYEE and DEPARTMENT whose columns are shown in Table 1. Exactly 60000 records are stored in table EMPLOYEE, where each record takes 30 bytes approximately, and only 10 records are stored in table DEPARTMENT, where each record takes about 10 bytes. For SQLite, we have used the default page size, which is 1024 bytes, and set up the number of buffer pages as 100.

For the experiments, we have arranged the records of table EMPLOYEE such that for each RRT-tree that is built by the recursive SQL queries, all leaf nodes of the tree have the same depth and all non-leaf nodes of the tree have the same number of child nodes. We name the number of RRT-trees that are built by a specific query as tree_count, the depth of the RRT-trees as tree_depth, the number of child nodes of the non-leaf

---

4 For the depth of each RRT, each root node of RRT-trees takes 1, each child RRT of the root node takes 2, and so on and so forth.
nodes as \textit{tree_fanout}. The four types of queries are evaluated for the case of having tables of EMPLOYEE and DEPARTMENT as foundation tables from the viewpoint of the recursive SQL query. Against the two foundation tables, one or more RRT-trees are built and by varying \textit{tree_depth} and \textit{tree_fanout} of the RRT-trees, the queries for producing XML values are evaluated. For the performance evaluation, each query has been executed 10 times and the average of their execution times is listed as the result of the query, where the execution time does not include the time to list the query result on the screen.

The query of Fig. 10 illustrates a typical example of the recursive SQL queries that are used in the experiments. In the query of Fig. 10, to limit the depth of the RT-trees, a pseudo-column \textit{level} of the recursive SQL query of Oracle is used; for the pseudo-column, each root RT takes 1, each child RT of the root RT takes 2, and so on and so forth. All queries that are used in the experiments make XML values having the same structure as that of the XML values that can be generated by the query of Fig. 10.

```
SELECT XMLNEST("EMPLOYEE", XMLATTRIBUTES(employee_id as "id"),
XMLFOREST(name as "NAME", salary as "SALARY",
          supervisor_id as "MANAGER", department_name as "DEPT"),
XMLNESTSPEC("SUBORDINATES")) as result
FROM employee e, department d
WHERE e.department_id = d.department_id
START WITH e.employee_id in (10, 21)
CONNECT BY prior employee_id = e.supervisor_id and prior level < 4;
```

Fig. 10. A recursive SQL query using XMLNEST.

On Oracle 10g, we have evaluated both the two types of the XQuery queries and the nested SQL queries using SQL/XML publishing functions. Table 2 illustrates the execution time of the three types of the queries under very small numbers of RRTs. As Table 2 shows, the XQuery query using recursive user defined functions have taken too much time for the given configuration of the two foundation tables. For example, to generate an XML value from an RRT-tree of 40 RRTs (\textit{tree_depth} 4 and \textit{tree_fanout} 3), the XQuery query using nested FLWOR expressions and the nested SQL query using

\begin{enumerate}
\item For a specific RRT-tree, once the \textit{tree_depth} is \(d\) and the \textit{tree_fanout} is \(f\), the number of RRTs in the RRT-tree becomes \(d\) if \(f\) is 1 and becomes \((f^{d-1})/(f-1)\) if \(f\) is greater than or equal to 2.
\end{enumerate}
SQL/XML publishing functions have been executed in 0.039 seconds and 0.008 seconds, respectively. However, for the same condition, the XQuery query using recursive user defined functions has required 2223 seconds (that is, 37 minutes and 3 seconds). Therefore, in the following experiments, the execution time of the XQuery query using recursive user defined functions will not be included.

### Table 2. Evaluation of the two types of the XQuery queries and nested SQL queries on Oracle 10g.

<table>
<thead>
<tr>
<th>RRT count</th>
<th>tree_count/tree_depth/tree_fanout</th>
<th>XQuery queries using recursive user defined functions</th>
<th>XQuery queries using nested FLWOR expressions</th>
<th>Nested SQL queries using SQL/XML publishing functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1/2/3</td>
<td>222 sec</td>
<td>0.009 sec</td>
<td>0.001 sec</td>
</tr>
<tr>
<td>13</td>
<td>1/3/3</td>
<td>721 sec</td>
<td>0.016 sec</td>
<td>0.004 sec</td>
</tr>
<tr>
<td>31</td>
<td>1/3/5</td>
<td>1718 sec</td>
<td>0.032 sec</td>
<td>0.007 sec</td>
</tr>
<tr>
<td>40</td>
<td>1/4/3</td>
<td>2223 sec</td>
<td>0.039 sec</td>
<td>0.008 sec</td>
</tr>
</tbody>
</table>

Fig. 11 illustrates the execution times of the XQuery queries using nested FLWOR expressions and those of the nested SQL queries using SQL/XML publishing functions under various RRT counts.

As Fig. 11 illustrates, the two graphs are nearly proportional to RRT counts. Even though RRT counts are formulated by the combination of tree_count, tree_depth, and tree_fanout, the execution time has not been influenced by any specific factor of them. Note that, in the range of RRT counts of 1365 ~ 3906, slope of the graph for the
The execution time of the XQuery query is almost 77 times steeper than that of the nested SQL query. For example, to generate XML values from the RRT-trees of 1365 RRTs (3906 RRTs, respectively), the XQuery query and the nested SQL query have been executed in 2.671 seconds (16.527 seconds, respectively) and 0.104 seconds (0.285 seconds, respectively) respectively such that the execution time of the nested SQL query is 26 times (58 times, respectively) faster than that of the XQuery query. From Table 2 and Fig. 11, for the generation of recursive XML values from relational data, at least on Oracle 10g, it is clear that making XQuery queries using nested FLWOR expressions cannot be a good choice and making XQuery queries using recursive user defined functions should be definitely avoided.

The main features of the recursive SQL query that contains XMLNEST work as follows in our implementation on SQLite:

1. Find the start tuples of the transitive closure and then append <root_tid, parent_tid, tid, deleted_flag> to each one of them, where tid is the tuple’s tuple identifier, parent_tid is the tid of the parent tuple of the tuple, root_tid is the tid of the start tuple of the transitive closure that the tuple belongs to, and deleted_flag signifies whether the tuple is a NULL-valued RRT or not. Note that the tuple identifier is an ever increasing integer that starts from 0.

2. For each one of the found start tuples, register the tuple into the RRT-table (a temporary table that is implemented as a B-tree and has <root_tid ASC, parent_tid DESC, tid ASC> as the B-tree’s key such that the table keeps the RRTs in the breadth first search order from leaf nodes to the root node of each RRT-tree) and then find all the descendant tuples of the start tuple such that for each found descendant tuple, register the tuple into the RRT-table after augmenting the tuple with <root_tid, parent_tid, tid, deleted_flag>. Note that upon registering a tuple in the RRT-table, the RT_reference_conditions of the recursive SQL query is checked against the tuple such that once the tuple does not satisfy the condition the deleted_flag of the tuple is set to TRUE.

3. For building XML values, a hash table is used as follows: fetch each RRT from the RRT-table in the breadth first search order of RRTs from the leaf level to the root level of each RRT-tree such that provided that either the deleted_flag of the tuple is FALSE or the entry value for the hash entry of the tuple is not empty, after building
an XML value with both the column values of the tuple and the entry value for the
hash entry of the tuple (i.e., the XML value that is built against all the descendants
of the tuple), append the XML value to the entry value for the hash entry of the
tuple’s parent RRT.

(4) Return the XML values that are built for root nodes of the RRT-trees.

On SQLite, we have evaluated the recursive SQL queries using XMLNEST and the
nested SQL queries using SQL/XML publishing functions. Fig. 12 illustrates the result
under various RRT counts.

Fig. 12. Evaluation of SQL queries using either XMLNEST or SQL/XML publishing functions on
SQLite.

As Fig. 12 illustrates, the two graphs are nearly proportional to RRT counts. Note
that, in the range of RRT counts of 1365 ~ 3906, slope of the graph for the execution
time of the recursive SQL query is just 4 times steeper than that of the nested SQL query.
For example, to generate XML values from the RRT-trees of 1365 RRTs (3906 RRTs,
respectively), the recursive SQL query and the nested SQL query have been executed in
0.0421 seconds (0.1345 seconds, respectively) and 0.0221 seconds (0.0453 seconds,
respectively) respectively such that the execution time of the nested SQL query is just
1.90 times (2.97 times, respectively) faster than that of the recursive SQL query. Note
that around 70% of the execution time of the recursive SQL query using XMLNEST
goes into the execution of the recursive SQL query itself. For example, to generate result
tuples of 1365 RRTs (3906 RRTs, respectively), the recursive SQL query without
XMLNEST has been executed in 0.0299 seconds (0.0812 seconds, respectively). Therefore by optimizing the execution of the recursive SQL query, the difference between the slopes of the two graphs in Fig. 12 can be minimized.

Since our goal lies in the production of recursive XML values from relational data, our implementation of the SQL/XML publishing functions and XMLNEST produces XML values as text. Because of that, the production could be fairly simple compared with producing them as the XML data type. We have tried to convert the XML values that are produced by the queries on SQLite to the XML values of the XML data type by using the SQL function XMLPARSE at Oracle 10g. It was done in reasonable amount of time. For example, for the XML value produced for 3906 RRTs of Fig. 12, the conversion just took 0.0912 seconds. Even though the conversion takes about twice as much time as the nested SQL query needs for producing the XML value, the whole time is acceptable.

As far as generating recursive XML values from relational data is concerned, provided that the depth of the RRT-trees to be found is known and is not deep, our experiments show that it is the best choice to make nested SQL queries using SQL/XML publishing functions. However, in addition to the weak points on the depth of the RRT-trees, nested SQL queries using SQL/XML publishing functions cannot provide the following two functionalities: making XML values that have sibling XML elements in some specific order, and acquiring diverse aggregate values for the hierarchy groups of tuples. In the sequel, once the depth of the RRT-trees is not known or it is so deep, making recursive SQL queries using XMLNEST will be the unique reasonable solution for generating recursive XML values from relational data while supporting the two functionalities that cannot be supported by the scheme of the nested SQL queries.

5. CONCLUSIONS

Various queries can be devised to publish transitively connected relational tuples as recursive XML values. Since the result tuples of the queries formulate hierarchies, which we call RRT-trees, the count, depth, and fanout of the RRT-trees become determining factors in the number of the recursive result tuples, which we call RRTs, on the trees. Provided that both the depth of the RRT-trees and the number of RRTs in the RRT-trees
are limited to some small number, the XQuery query using nested FLWOR expressions and the nested SQL query using SQL/XML publishing functions could be very good querying schemes. Even in that case, the scheme of the XQuery query using recursive user defined functions could not be a solution because of its enormous amount of execution time. Once some reasonable number of RRTs are retrieved under some limited depth of the RRT-trees, the XQuery query using nested FLWOR expressions is determined not to be a good choice and the nested SQL query using SQL/XML publishing functions becomes an excellent scheme.

Provided that the hierarchies become so deep or the depth of them is not known in advance, making nested SQL queries using SQL/XML publishing functions is either hard or impossible. To solve that problem, we have proposed an SQL aggregate function XMLNEST along with the notion of the hierarchy group and the NULL-valued RRTs. Both the recursive structure and the order of sibling XML elements of the XML values can be specified in the invocation of XMLNEST; and the traditional aggregate values for subtrees of RRT-trees can be obtained through the XMLNEST-related functions that are used as arguments of XMLNEST. The scheme of the recursive SQL queries using XMLNEST can resolve the problem of the nested SQL queries, and queries of the proposed scheme have been executed in time with just about 43% overhead to the execution time of the recursive SQL queries without XMLNEST.

REFERENCES


21. SQLite, Available at: http://www.SQLite.org.

Sung-Chul Park (朴成喆) received the B.S., M.S. and Ph.D. degrees in Computer Science from Kyungpook National University, Daegu, Korea, in 1996, 1998 and 2010. In 2010, he joined the faculty of Computer Science of Young Jin College, Daegu, Korea. He is currently a Full Time Lecturer in the Department of Computer Science of Young Jin College. His research interests include storage systems, query optimization, and XML databases.

Young-Chul Park (朴永鐵) received the B.S. and M.S. degrees in Electrical Engineering from Seoul National University, Seoul, Korea, in 1977 and 1981. He also received the M.S. and Ph.D. degrees in Computer Science from Northwestern University, Evanston, U.S.A., in 1986 and 1989. From 1984 to 1993, he has been a Professor at the Department of Computer Science of University of Ulsan, Ulsan, Korea. In 1993, he joined the faculty of Computer Science of Kyungpook National University (KNU), Daegu, Korea. He is working as a Professor in the School of Computer at the KNU. His research interests include transaction management, storage systems, query optimization, XML databases, and spatial databases.