Hierarchical data management in relational systems.

MALIKOV A., GULEVSKIY Y., PARKHOMENKO D., MALIKOVA E.
Information Systems and Technologies Department
North Caucasus State Technical University
Stavropol, Kulakova st. 2, 355028
THE RUSSIAN FEDERATION
malikov@ncstu.ru, mitix@ncstu.ru

Abstract: Methods for hierarchical data management are suggested. Methods for optimizing data bases with hierarchical data are suggested and analyzed. Interface model between the database, containing hierarchical structure and the end-user software is suggested.

Key-Words: Hierarchical data structures, materialized path, optimization, data accessing.

1 Introduction
For years, effective data management has been associated with relational data model. It is a reasonable fact – relational model is simple, descriptive and possesses a developed mathematical apparatus, the most advanced in comparison with other mathematical theories of data abstraction. One of the problems of relational model is complexity of hierarchical data management. In this context following typical problems take place: representation of hierarchies as two-dimensional tables is not intuitive; use of data manipulation languages is impeded; data base queries for such data show low performance.

Generally, hierarchical data is not to be understood only as hierarchy as it appears in the real world, like “employer-employee” relationship. The idea of hierarchy can be extended to cover the whole application domain, which expert is creating a database for. Such representation is more universal in comparison with two-dimensional tables. There are a lot of papers [1][2][3][4], concerning problems of modeling and implementing hierarchies. Large RDBMS manufacturers have implemented new features for managing hierarchical data in latest versions of their products. For example, Microsoft in SQL Server 2008 introduced a new way of managing tree structures – a new data type named HierarchyId. HierarchyId implements the idea of materialized path in a tree [1][5][6][7], which considerably simplifies hierarchical data management. Attributes of HierarchyId type in a table are indexed using clustered index, which dramatically increases data selection performance. Length of HierarchyId value depends on several tree characteristics, firstly on an average number of child nodes on a particular level of the tree.

Problems of materialized path systems, HierarchyId in particular are:

1. Operations, leading to materialized path recalculation, for example subtree movement.
2. Problems of tree management in distributed systems.
3. Access control problems.
4. Hierarchical key length limit.

Anyway, in spite of these problems, new technologies in MS SQL Server 2008 are very important.

2 Methods of modeling a directed graph using “materialized path” concept
There exist at least 3 ways of managing hierarchical data in relational systems:
1. Using “adjacency list” model.[8]
2. Using XML data.
3. Using “materialized path” concept – (HierarchyId and similar implementations).

If a hierarchy is represented as a directed graph, it is easy to use “adjacency list” model, but there occur several problems, connected with low selection performance. Also, it is impossible to use XML and “materialized path” as they are, without transforming a directed graph into a tree.

One of the ways of modeling an acyclic directed graph is to use special methods of node traversal to create a hierarchical key [3,5]. Let’s generalize these methods to represent arbitrary acyclic directed graphs as trees and create a “materialized path” key to encode graph nodes. As a simple example, let’s take a hierarchy in a company, where each employee can have several junior employees and at the same time can be a junior employee for several (0, 1 or more) managers. Let’s represent this hierarchy of employees as a directed network, where 0 level nodes have no parents, and create a relational table to store it in the database:
CREATE TABLE Organization
(employeeId int NOT NULL,
treeKey varchar(900),
typeLink bit,
level int)
Field names stand for:
employeeId – identification number of an employee;
treeKey – stores materialized path value;
linkType – flag, indicating whether the link between
nodes in the tree matches corresponding existing link
in the graph or the link was added after
transformation of graph into the tree;
level – level number of the node in a directed
network.

treeKey field is indexed using clustered index.
Varchar(900) data type is used for demonstrativeness
and has the same meaning as HierarchyId.

Let's define a binary relation \( o \) – traversal order
of the directed network, it defines links between
nodes in a form of a forest. This relation is
antisymmetrical, transitive, and reflexive.

Algorithm for the directed network traversal
consists of the following steps:

1. On the minimal level of the directed network
choose the leftmost node \( u \) which has links with
parent nodes.
2. Let \( \{ v_1, v_2, \ldots, v_m \} \) be set of ancestors of \( u \). To
keep antisymmetric property, \( v_1 \circ v_2 \circ \ldots \circ v_m \) relation is defined, such that if \( i, j \in 1 \ldots m \) and
\( \exists (v_i \circ v_j) \) then \( v_j \circ v_i \) relation is substituted by
\( v_i \circ v_j \) relation. Each node \( v_i \) with an empty
treeKey value is assigned
\[
treeKey_{v_i} = \text{right}((\text{replicate}(a, l) + ID_{v_i})_l) + \ldots +
\text{right}((\text{replicate}(a, l) + ID_{v_i})_l), \quad \text{typeLink}_{v_i} = 0 \ ,
\]
\( i \in 1 \ldots m \) where \( treeKey_{v_i} \) is \( treeKey \) of node \( v_i \), \( a \)
– separator character, \( l \) – length of the key part to
encode a node, \( ID_{v_i} \) – \( v_i \) node’s identifier, "+" –
string concatenation operator, \( \text{replicate}(x_1, x_2) \) –
function, returning a character string of \( x_1 \) characters
repeated \( x_2 \) times, \( \text{right}(x_1, x_2) \) – function, which
returns \( x_2 \) rightmost characters of \( x_1 \) string.
3. \( \circ \) relation is defined: \( v_1 \circ v_2 \circ \ldots \circ v_m \circ u \) ;
node \( u \) is assigned
treeKey_{v_i} = treeKey_{v_i} + \text{right}((\text{replicate}(a, l) + ID_{v_i})_l)
and typeLink_{v_i} = 1 \ flag.
4. Choose next working node, right of previous
(already processed) node. If current level doesn’t
contain unprocessed nodes, go to next level.

5. If directed network contains no more
unprocessed nodes, then stop, else go to step 3.

After running this algorithm, relational table is
created, containing a tree, transformed from a graph
(from now on, \textit{transformed tree}), with as much
linkType values as the number of nodes in the
network, with parent links present.

Consider the following hierarchy, where node
labels are employeeId values:

Picture 1. Example of a directed network.
Table 1. Contents of “Organization” relational table.

<table>
<thead>
<tr>
<th>employeeId</th>
<th>treeKey</th>
<th>linkType</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>124</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1243</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>12435</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>124356</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>357</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Now we can use the benefits of “materialized
path” keys. Consider a database query, which selects
all junior employees for an employee with
employeeId = 1:

\[
\text{Select } b.* \text{ from Organization } a, \text{ Organization } b \text{ where } a.employeeId=1 \text{ and } b.linkType=1 \text{ and } a.treeKey<b.treeKey \text{ and } b.treeKey<a.treeKey+‘a’
\]

Query results are listed in Table 2.

Table 2. Query results.

<table>
<thead>
<tr>
<th>employeeId</th>
<th>treeKey</th>
<th>linkType</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>124</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>124356</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

This approach is suitable only for small directed
graphs, because relation for storing arcs of the tree
can have rather significant size, while “materialized
path” key length can exceed RDBMS' limit for
indexed fields. In fact, if a materialized path key
contains information about all ancestors of the node,
including the node itself, then key length \( L \) of an
arbitrary node on the \( l \) level of directed network is
equal to:

\[
L = A \cdot B = \frac{a^l - 1}{a - 1} \cdot \text{ceil}(\log_2 b),
\]
where \( A = \frac{a' - 1}{a - 1} \) – is the number of node’s ancestors;
\( B = \text{ceil}(\log_2 b) \) – key part length to encode information about one node;
\( a > 1 \) – average number of nodes’ parents in the directed network;
\( b \in (1, \infty) \) – average number on nodes’ children in the directed network;
\( \text{ceil}(x) \) – function, returning minimal integer number, greater than the argument.

Approximate key length can be estimated using formula: \( L = l a^{i-1} \cdot \text{ceil}(\log_2 b) \)

3 Optimizing transformed tree relation size.

Transformed tree relation size can be decreased. Different graph traversal algorithms result in different tree configurations. Trees, converted from the same graph may vary in the number of nodes, number of links and as a result, time of query execution.

We suggest an optimization method for a transformed tree. Let’s take minimum number of nodes as a criterion of optimization.

Consider a graph of atomic values of the application domain. This graph is a transformation of relational tables graph. At the very beginning, initial atomic value set is empty and database administrator must define relational table traversal order to create an atomic value tree.

Assume that atomic value tree is defined fully or database administrator is aware of data distribution trends in the application domain. This information in essential to optimize the tree structure using node number minimalization criteria. In accordance with relational theory, full set of atomic values of application domain can be represented as a two-dimensional relational table in the first normal form (1NF). This table is associated with an atomic value tree.

Let \((A_i, x, A_{i+1}, y)\) be ordered set of atomic values, where each atomic value \( y \) appertains to attribute \( A_{i+1} \), and each atomic value \( x \) appertains to attributes \( A_i \), which stand before \( A_{i+1} \) in the sorting order, \( i, i+1 \in [1, I] \), where \( I \) is the cardinal number of the relation. Let \( N_{j,i+1} \) be number of unique ordered sets \((A_i, x, A_{i+1}, y)\). Exact solution of this problem is to find such repositioning of attributes \( \{A_1, A_2, \ldots, A_I\} \), that \( \sum_j N_{j,i+1} \rightarrow \min \), where \( j \) - index of the actual repositioning. Computational complexity of this solution can be estimated as \( I! \).

To find an approximate solution for this problem, let’s introduce attribute selectivity concept \( S_{A_i} = \frac{K_i}{K} \), where \( K_i \) is the number of unique values of \( A_i \) attribute, \( K \) – is the arity of 1NF relation. Thus, solution of this problem can be stated as selectivity calculation of each attribute, and sorting of these attributes according to their calculated selectivity. If we ignore sorting algorithm, computational complexity of this solution is estimated as a linear polynomial of \( I \).

Contents of Table 3 can be represented as a directed graph, shown in Picture 2, and as Table 4, which defines “parent-child” relationship between application domain objects.

Table 3. 1NF relation, which contains information about employees.

<table>
<thead>
<tr>
<th>Organization</th>
<th>Department</th>
<th>Position</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mega DB Inc.</td>
<td>Software</td>
<td>programmer</td>
<td>Waters R</td>
</tr>
<tr>
<td></td>
<td>department</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mega DB Inc.</td>
<td>Software</td>
<td>programmer</td>
<td>Hammet K</td>
</tr>
<tr>
<td></td>
<td>department</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mega DB Inc.</td>
<td>Software</td>
<td>manager</td>
<td>Smith J</td>
</tr>
<tr>
<td></td>
<td>department</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Picture 2. Directed graph of application domain objects.

Table 4. “Parent-child” relationship between application domain objects.

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mega DB Inc.</td>
<td>Waters R</td>
</tr>
<tr>
<td></td>
<td>Hammet K</td>
</tr>
<tr>
<td></td>
<td>Smith J</td>
</tr>
<tr>
<td>Software</td>
<td>Waters R</td>
</tr>
<tr>
<td></td>
<td>Hammet K</td>
</tr>
<tr>
<td></td>
<td>Smith J</td>
</tr>
<tr>
<td>programmer</td>
<td>Waters R</td>
</tr>
<tr>
<td></td>
<td>Hammet K</td>
</tr>
<tr>
<td>Manager</td>
<td>Smith J</td>
</tr>
</tbody>
</table>

Attributes in Table 3 \{Organization, Department, Position, Employee\} are sorted according to their selectivity \( \{1, 1, 2, 3\} \). This information is
essential for node traversal algorithm. Execution results of this algorithm are shown in Picture 3 and Table 5.

![Picture 3](image)

Picture 3. Links between application domain objects in a form of a tree.

Table 5. Using materialized path to represent a hierarchy of application domain’s objects.

<table>
<thead>
<tr>
<th>Object ID</th>
<th>Value</th>
<th>treeKey</th>
<th>typeLink</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mega DB Inc.</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Software department</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>programmer</td>
<td>123</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>manager</td>
<td>124</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Waters R.</td>
<td>1235</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Hammet K.</td>
<td>1236</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Smith J.</td>
<td>1247</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4 Relational explanation of “materialized path” concept

In general, number of vertexes and number of arcs in a directed graph differ. Thus, in contrast to trees, to define a directed graph two relational tables must be used, instead of one. First relation contains node information, while the second one contains arc information. Let us denote “node relation” cardinality $n$ and “arc relation” cardinality $m$.

“Parent-child” and “Materialized path” concepts can be combined to gain advantages from both of them. “Parent-child” concept defines immediate reachability relationship on the set of vertexes. Reachability relationship on the directed graph is a reflexive-transitive closure of immediate reachability relationship. This relationship is reflexive, transitive, antisymmetric and can be represented as a square reachability matrix. To implement such a matrix, two relational tables need to be created: first one contains vertex data, the second one contains vertex reachability data as \{ancestor, descendant\} couples. Let’s call this approach an “ancestors-descendants” concept.

Relational table, containing vertex data looks like:

```
CREATE TABLE HierarchyObject
(ID int NOT NULL primary key, -- vertex ID
 name varchar(50), -- vertex name
 level int -- vertex level in the directed network)
```

Relational table, containing vertex reachability data looks like:

```
CREATE TABLE HierarchyLink
(IDancestor int NOT NULL, -- ancestor vertex ID
 IDdescendant int NOT NULL -- descendant vertex ID)
```

HierarchyLink.IDancestor and HierarchyLink.IDdescendant attributes are foreign keys to HierarchyObject.ID attribute. Most database queries to HierarchyLink table will contain filters on IDancestor and/or IDdescendant attributes, therefore both these attributes are indexed using a non-clustered b-tree index. It is recommended to use a clustered index on the attribute with low selectivity, to increase performance.

Suggested approach allows defining of “materialized path” in more traditional (as for relational systems) way: as a finite set of tuples. Picture 4 shows the comparison between “materialized path” and “ancestor-descendant” concepts.

![Picture 4](image)

Picture 4. Different ways to represent a hierarchy
a) graphical view b) “ancestor-descendant” concept c) “materialized path” concept.

HierarchyLink table is based on vertex reachability matrix, shown in Table 6.

Table 6. Reachability matrix for tree’s vertexes.

<table>
<thead>
<tr>
<th>vertexes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Let’s estimate HierarchyObject and HierarchyLink arity. HierarchyObject’s arity is equal to the number of vertexes in the graph - \( n \). HierarchyLink’s arity \( H \) is comparable to the number of cells in the reachability matrix: \( H \leq n^2 \).

Reachability matrix is antisymmetric, therefore HierarchyLink’s arity can be estimated as area of a right triangle, under the main diagonal of reachability matrix: \( H \leq \frac{n^2 + n}{2} \). This estimation is quite rough, because of the undetermined fill-factor of the triangle.

Number of reachable vertexes for an arbitrary vertex on the \( l \)’th level is estimated as \( A = \left( a^0 + a^1 + a^2 + \ldots + a^{l-1} \right) \leq la^{l-1} \). If \( a = 1 \), then directed network is a tree and \( A = l \). If \( a \in (1, \infty) \) then \( A = \frac{a^l}{a - 1} < n \). If directed network consists of \( l \) levels, then HierarchyLink’s arity is estimated using formula: \( H \leq \frac{a^l}{a - 1} \cdot n \leq la^{l-1}n \). In case of a tree, the following statement is true: \( a = 1 \), \( H \leq l \cdot n \).

5 Directed graph data manipulation

Consider several examples, which show some peculiarities of creating SQL queries to HierarchyObject and HierarchyLink.

Problem 1. Find shared ancestors of \(@ID1\), \(@ID2\), \(@ID3\) vertexes:

\[
\text{Select } hl3.* \text{ from HierarchyLink } hl1, \text{ HierarchyLink } hl2, \text{ HierarchyLink } hl3 \text{ where } \begin{align*}
hl1.IDdescendant &= @ID1 \\
 hl2.IDdescendant &= @ID2 \\
 hl3.IDdescendant &= @ID3
\end{align*} \quad \text{and} \quad \begin{align*}
 hl1.IDancestor &= hl2.IDancestor & \text{and} & \quad hl2,IDancestor &= hl3.IDancestor \\
 hl1.IDancestor &= @IDchild & \text{and} & \quad hl2,IDancestor &= @IDparent
\end{align*}
\]

Problem 2. Find shared descendants for \(@ID1\), \(@ID2\) vertexes, which are not descendants for \(@ID3\) vertex:

\[
\text{Select } hl3.* \text{ from HierarchyLink } hl1, \text{ HierarchyLink } hl2, \text{ HierarchyLink } hl3 \text{ where } \begin{align*}
 hl1.IDancestor &= @IDchild \\
 hl2.IDancestor &= @IDparent \quad \text{and} & \quad hl3.IDancestor &= @ID3 \\
 hl1.IDdescendant &= @IDchild \quad \text{and} & \quad hl2,IDdescendant &= @IDparent \\
 hl1.IDdescendant &= hl2,IDdescendant \quad \text{and} & \quad hl2,IDdescendant &= hl3.IDdescendant
\end{align*}
\]

Selection queries from HierarchyObject and HierarchyLink tables show demonstrativeness, high performance and remain constant event when graph structure changes.

Arc addition, deletion and modification operations involve processing of a subset of HierarchyLink table records. The most time-consuming operation is deletion of an arc, which involve deletion of ancestor information for every descendant node. If HierarchyLink contains only IDancestor and IDdescendant attributes, it is not possible to determine which records must be deleted. This limitation is a result of the fact, that there can exist alternative paths between a couple of vertexes.

There are at least 2 ways to solve this problem:

1. Add new attributes to HierarchyLink, showing the nature of connections between vertexes. For example, a flag, indicating whether the link exists between a couple of vertexes. Then, to delete or modify the arc, we need to find recursively all linked vertexes, to determine their subset, which will be deleted by the query. This approach is not acceptable, because it leads to performance decrease.

2. Convert the source graph to a directed graph, without alternative paths between vertexes. Such modification can be made using special algorithms, implemented in databases, normalized on the basis of selection and join operations [4].

Consider peculiarities of the database queries, performing addition, deletion and modification of an arc in the directed graph, which has no alternative paths between couples of vertexes. Source information for these queries can be obtained from \{@@IDparent, @@IDchild\} values, which identify an actual arc.

Problem 3. Add \{@@IDparent, @@IDchild\} arc to the directed graph:

\[
\text{Insert into HierarchyLink select } b.IDancestor, \text{ a.IDdescendant from HierarchyLink } a, \text{ HierarchyLink } b \text{ where } a.IDancestor=@@IDchild \quad \text{and} \quad b.IDdescendant=@@IDparent
\]

Select statement returns all couples among descendants of @IDchild vertex including the vertex itself, and ancestors of @IDparent vertex, including the vertex itself.

Problem 4. Delete \{@@IDparent, @@IDchild\} arc from the graph:

\[
\text{Delete from HierarchyLink where exists (select * \text{ from HierarchyLink } a, \text{ HierarchyLink } b \text{ where } a.IDancestor=@@IDchild \quad \text{and} \quad b.IDdescendant=@@IDparent \quad \text{and} \quad HierarchyLink.IDancestor=b.IDancestor \quad \text{and} \quad HierarchyLink.IDdescendant=a.IDdescendant)}
\]

Problem 5. Replace \{@@IDparentOld, @@IDchild\} arc by the \{@@IDparentNew, @@IDchild\} arc:

\[
\text{Delete from HierarchyLink where exists (select * \text{ from HierarchyLink } a, \text{ HierarchyLink } b \text{ where } a.IDancestor=@@IDchild \quad \text{and} \quad b.IDdescendant=@@IDparent \quad \text{and} \quad HierarchyLink.IDancestor=b.IDancestor \quad \text{and} \quad HierarchyLink.IDdescendant=a.IDdescendant)}
\]

\[
\text{Insert into HierarchyLink select } b.IDancestor, \text{ a.IDdescendant from HierarchyLink } a, \text{ HierarchyLink } b \text{ where } a.IDancestor=@@IDchild \quad \text{and} \quad b.IDdescendant=@@IDparentNew
\]
6 Interface between the database, containing hierarchical data and the end-user software

Regardless of the hierarchical structure implementation method, one needs software to manage data effectively. There exist different approaches to create such software. Mainly, they differ in the way of implementation of hierarchical structure in the database.

Interface between such a database and the end-user software includes the following levels:

1. Database level, where hierarchical structure is defined.
2. Data access layer (DAL).
3. Software object model.

Software object model is a set of classes (in terms of an object-oriented programming language), created to represent an in-memory graph data structure. These classes encapsulate information about a particular entity of an application domain, about atomic values etc.

Data access layer – is a level of abstraction between relational representation of hierarchical and software object model. It is a piece of software, which is aware of data base specifics and is used to provide software object model with actual data.

Data access layer’s features are the following:

1. Abstracting from implementation of hierarchical structure.
2. Caching a subset of atomic values.
3. Adaptation of the client software to the changes in the application domain.

When you need to use another hierarchical structure implementation, data access layer becomes very important, because it allows to adapt your system very fast, with minimal changes in the code.

Also DAL increases the overall stability of the system, in case of changes in the application domain, because addition and deletion of entities doesn’t lead to link changes between the objects, while in relational systems database refactoring leads to changes in the query code.

7 Conclusions

The most popular methods to define a tree structure in a relational database are: “parent-child” concept, XML data type, «materialized path» concept. These methods are effective to use with trees, but do not allow effective management for data, represented as directed graphs. To define a directed graph in a relational database, these methods can be modified. One of the possible ways is a transformation of a directed graph into a tree, which is represented as a two-dimensional relational table, with the help of a special “materialized path” attribute. This approach allows using of “materialized path” benefits, but is very sensible to the source graph configuration, when tree table’s arity and/or materialized path key’s length are increased dramatically.

Use of a reachability matrix in a relational system can be considered promising. To manage a set of graph’s vertexes, a special relational table is created, which defines a reachability relationship between couples of vertexes. The above suggested approach allows to implement “materialized path” concept in a more traditional (as for relational systems) way: as a finite set or relational records. Query examples, given above, show high performance.

Results of this paper are implemented as a special software program – «BiZone» hierarchical data management system, which is used by a number of national and federal institutions in the Russian Federation.

References:


