Tree Denormalization

Copyright (C) 2002 by Thomas Beck. All rights reserved.

INTENT

Avoid iterations when querying tree related data.

ALSO KNOWN AS

Hierarchy Denormalization

MOTIVATION

When modeled in a normal form, finding all the direct and indirect childs of a node requires at least as many iterations as the distance between the nodes. Some SQL dialects allow it to be done it implicitly in one statement, but the performance is anyway impacted.
The goal of the denormalization is to store all the possible direct and indirect links (all the possible paths) in order to avoid any iteration or navigation through the tree and take full advantage of the relational power, allowing joins on this structure.

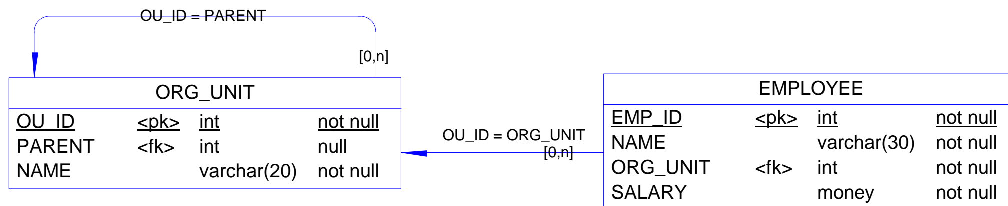
APPLICABILITY

Use the Tree Denormalization Pattern when:

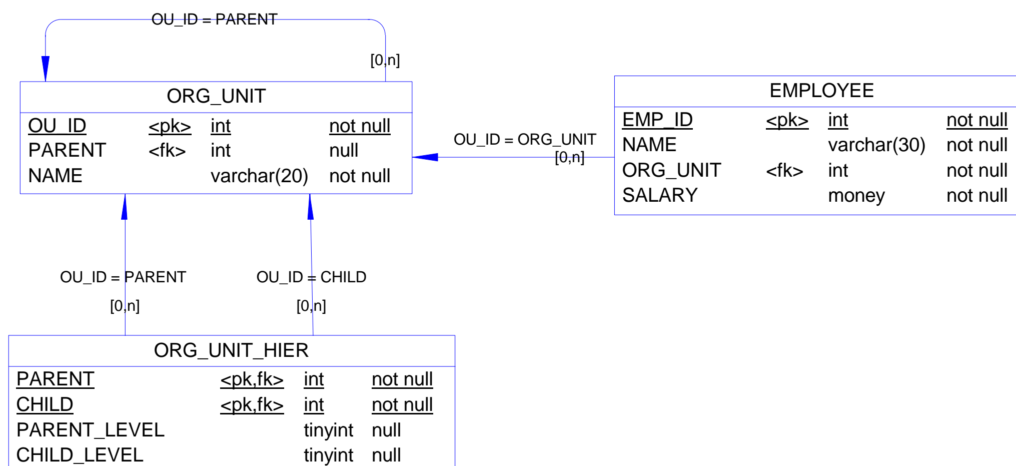
- The tree is not updated frequently and the update performance is not an issue.
- There is enough storage space to store the current and future expanded graph structures.

STRUCTURE

Example of normalized structure:



Example with the denormalized table added:



CONSEQUENCES

- It is possible to find out all the nodes below a given node with a “one-shot” query.
  Exemple:
  `SELECT CHILD FROM ORG_UNIT_HIER WHERE PARENT = x`
  It is possible to join such a list of nodes with other tables.

IMPLEMENTATION

Fields of the denormalized table:

PARENT: parent node
CHILD : child node
PARENT_LEVEL : level of the parent node. Ex. 1 for the root level, 2 for the root childs, etc.
CHILD_LEVEL: level of the child node.

Suppose the following contents in the ORG_UNIT table:

```
OU_ID       PARENT      NAME
----------- ----------- --------------------
1           NULL        The Company
2           1           Financial
3           1           Admin
4           1           IT
5           4           Support
6           5           Development
```

Then, the contents of the denormalized ORG_UNIT_HIER table are the following:

```
PARENT      CHILD       PARENT_LEVEL CHILD_LEVEL
----------- ----------- ------------ -----------
1           1           0            0
1           2           0            1
1           3           0            1
1           4           0            1
1           5           0            2
1           6           0            3
2           2           1            1
3           3           1            1
4           4           1            1
4           5           1            2
4           6           1            3
```

Note that each node contains also a self-reference (PARENT=CHILD). That's very useful for most queries, because if we want for example to know the total amount of salaries in IT we simply query:

```
SELECT SUM(E.SALARY)
FROM   EMPLOYEE E
,      ORG_UNIT_HIER H
WHERE  H.CHILD = E.ORG_UNIT
AND    H.PARENT = 4
```

And that includes the node 'IT' itself.

PARENT_LEVEL and CHILD_LEVEL fields are not mandatory, but quite useful. We can for example query only the direct childs of a node (WHERE CHILD_LEVEL = PARENT_LEVEL+1)

Changes over time

Of course, most hierarchies change over time, and the historical status of the tree is functional as well. The same technique used un the "History of change" pattern can be applied, adding START and END instants, except that instead of just versioning tuples

we can directly version the entire contents of the ORG_UNIT_HIER table. That means that each time there is a change in the tree, we create a new complete version of the tree and explose it in the denormalized table, with the couple START,END representing the version number.

Sorting

A common requirement is to be able to retrieve a tree structure in a given order. That can be done by adding an ORDER_ID field to the ORG_UNIT_HIER table.
This field will simply be used in the ORDER BY clause.

SAMPLE CODE

Example of code populating the ORG_UNIT_HIER table (here in Sybase/Miscrosoft T-SQL):

```
declare @level  tinyint        select  @level = 1

begin tran populate_org_unit_hier

   --  empty the table
   delete  ORG_UNIT_HIER

   -- insert the root record, as self-reference
   insert into ORG_UNIT_HIER (PARENT, CHILD, PARENT_LEVEL, CHILD_LEVEL)
   select  OU_ID, OU_ID, 0, 0
   from    ORG_UNIT
   where   PARENT is null

   -- insert subsequent levels
   while   1=1
   begin
      -- insert the new level as direct childs
      insert  into ORG_UNIT_HIER (PARENT, CHILD, PARENT_LEVEL, CHILD_LEVEL)
      select  h.PARENT, o.OU_ID, h.PARENT_LEVEL, @level
      from    ORG_UNIT_HIER h  -- all the IDs with childs in the previous level
      ,       ORG_UNIT o       -- the level to be added
      where
      ------- define h   ----------------------
              h.CHILD_LEVEL = @level -1
      ------- define o   ----------------------
      and     o.PARENT = h.CHILD

      if @@rowcount = 0 break  -- end of loop condition

      -- insert the new level as self-reference
      insert  into ORG_UNIT_HIER (PARENT, CHILD, PARENT_LEVEL, CHILD_LEVEL)
      select  CHILD, CHILD, CHILD_LEVEL, CHILD_LEVEL
      from    ORG_UNIT_HIER     -- all the IDs just inserted
      where   CHILD_LEVEL = @level

      select  @level = @level + 1
  end -- while
```

```
commit  populate_org_unit_hier

update statistcs ORG_UNIT_HIER
```

KNOWN USES

Tree structures. Can be generalized also to some other types of graph.

RELATED PATTERNS

History of Change.