Genealogical Representation of Trees in Databases
First Draft

Miguel Sofer <mig@utdt.edu>
Universidad Torcuato Di Tella
Buenos Aires
Argentina

May 6, 2000

Abstract

blah blah ...

1 Introduction

Trees are a very frequent data structure. They are the natural representation for instance for organizational charts, threaded discussion groups, some bills of materials, ...

At least two alternative representations for trees in RDBMs are known and used:

1. **Pointers:** a field in the child record references the parent node. This seems to be the canonical representation. Some DB engines provide special SQL extensions to simplify tree searches; Oracle tree extensions are an example (see for instance [1]); DB2's WITH can be used for this purpose too (see [3], pp 139-162).

2. **Nested Sets:** two numeric fields in every node record code the tree structure. I can't provide a better or briefer description of this method than the four articles [2].

These two methods offer different advantages and disadvantages:

- Pointers are extremely efficient for node insertion and/or deletion, but require recursive table accesses to search the tree (I do not know the implementation details of the Oracle tree extensions, which as far as I know may solve this problem internally; they definitely solve it for the end user).

- Nested sets are very efficient for tree searches, but are rather expensive for node insertion and/or deletion: they require updating potentially many nodes.

We propose here a different representation, based on node identifiers which are "genealogical identifiers": they contain the complete genealogy of the node, i.e., the list of ancestors up to the root of the tree.

This allows to replace many searches in database tables with string operations on the index. The result, as explained in Section 3 is that tree searches proceed at "nested sets" speed, while node insertions and deletions are as fast as with pointers.

The obvious downside of the method is that the primary key in the tree needs to be a variable size text field, and that the identifiers may be extremelly long for deep trees. We will provide estimates of the size required as a function of the magnitude of the tree.

2 Genealogical identifiers for trees

2.1 Definition

We define *genealogical identifiers* recursively as follows:

> **Definition:** *The genealogical identifier (gID) of a node is obtained by appending a child identifier to the genealogical identifier of the parent node.*

Remark that genealogical identifiers are rather well known and used; common examples are the "path+file-name" in a computer file system and the URLs within a WWW.

The name "genealogical identifier" is suggested by the fact that the value of the identifier contains the complete genealogy of the node: it contains as a substring the gID of its father, which in turn contains as a substring the gID of the grandfather, . . .

The root node of the tree has a gID with value '' (the empty string), as it has no parent.

2.2 Child identifiers

The obvious child identifier is a zero-based counter: identify the child by the number of older brethren it has.

We could represent the counter in base 10; this however is extremely wasteful of resources. It is much better to represent the counter in as large a base as possible: interpret as numbers a set of characters larger than {0,1,. . . 9}.

As tree operations will involve string operations on the indices, in order to avoid a "quoting hell" it is desirable to avoid using any character with a special meaning in LIKE expressions or regular expressions; i.e., we will not use any of the symbols `. * ^ \ [ ] { } ( ) < >` ? | & $

We propose to reserve also / as a separator (see "Variable Sized gID" below).

If we limit ourselves to ascii characters, and avoid to be safe a lot of other characters, we can use numbers in base 64 by representing

- 0-9 with '0'-'9' (dec ascii code 48-57)
- 10 with ':' (dec ascii code 58)
- 11 with ';' (dec ascii code 59)
- 12-37 with 'A'-'Z' (dec ascii code 65-90)
- 38-63 with 'a'-'z' (dec ascii code 97-122)

By using base 64, up to 4096 children can be represented using two such digits, up to 262144 with three digits, and up to 16777216 with four digits.

If the RDBMs supports international characters, it is possible to further increase the base; as an example, by using the 95 additional characters of the latin-1 character set, we could code numbers in a base up to 160 – remark that every single digit is still one byte in this representation. This means that we expand the symbols above by representing

- 64-159 with dec latin1 code 160-255

In base 160, up to 25600 children can be represented using two digits, up to 4096000 with three digits, and up to 6.5E+08 with four digits.

Remark that base conversions only need to be performed at insertion time, when the index of a new node is computed. They will therefore only have an impact on insertion timings.

2.3 Counters: "delimited" vs. "fixed size"

The standard representation of gID uses a variable size child identifier, and delimiters to separate the gID of the child node from the gID of its parent. For example, we can represent the fifth child of node '/23/27/1' as '/23/27/1/4'. Let us call this a **vgID** representation (Variable Size Genealogical ID).

This representation allows for any number of children of a node, subject only to the limitations the RDBMS may have as to the length of a variable sized string.

Alternatively, we could choose to limit from the outset the quantity of children that a node may have; this limit would depend of course on the application. Let us call this a **fgID** representation.

For example, if no node is allowed to have more than 25600 children, we could represent the counters always with 2 digits. The node which was previously '/23/27/1/4' is now '23270104'. If we require a three digit representation of nodes (up to about 4 million children), then it will be represented as '023027001004'.

2.4 Ordering of nodes

For some applications it is necessary to obtain subtrees ordered according to some special rules. For instance:

1. the complete subtree starting at a node is listed immediately after the node in question ("depth first")
2. nodes with a common parent are listed chronologically

For instance, the display of an organization chart is usually required to satisfy at least the first condition. In a threaded discussion group one wishes to satisfy both conditions to display the messages in a thread – the threads themselves (i.e., children of the root node) are usually listed in inverse chronolical order.

To make a particular ordering efficient, it would be a nice feature if it could be made to coincide with a lexicographic ordering of the indices –i.e., as produced by an "ORDER BY id ASC" in SQL. The lexicographic ordering of fgID satisfies both conditions. The lexicographic ordering of vgID as described above satisfies the first requisite if the separator has the minimal binary representation of all allowed symbols in an index – this is why we reserved / for the separator. But the second property is missing: for instance, the index '/1/10' is lexicographically before '/1/2'.

If the second property is also required for vgID, we can specify the child identifiers with counters built in the following way: represent a number by a string of digits, where

- the first digit D_0 represents the length in digits of the decimal expansion of the number, minus one
- the following $(D_0 + 1)$ digits are the decimal expansion of the number

Let us call these identifiers **m-vgID**, "m" for modified.

As an example, the node which was previously represented by /15/3/182 will, after this modification, have the index /115/03/2182.

The lexicographic ordering of m-vgID is the desired ordering of the tree nodes. The cost of this property is that (a) the ID are now longer, (b) no node can have more than 160^{160} children (actually, this is a non-issue), and (c) the index structure is redundant, some formally correct indices are invalid –e.g., /316/013/11. The third issue can be addressed by keeping a strict control on the generation of new indices to insure that all indices are formally correct.

The issue of the reverse chronological indexing of threads in threaded discussion groups can be addressed easily enough in fgID: count "down" instead of "up" the children of the root node – this implies only an inconsequential modification of the node insertion routine, as shown below. The problem is less trivial with vgID; in this case, maybe a thread identifier should be kept in a different field - i.e., representing the structure as a forest rather than a tree, where the thread_id field selects the "tree" in the forest.

3 Tree operations using genealogical indices

In this section we show how to implement various tree operations using gID as the primary key in the node table.

Some implementation issues are relevant here, especially concerning the utilisation of indices by the DB engine.

We discuss a tree represented in a table of the form

```
CREATE TABLE tree (
   gid       text PRIMARY KEY,
   nchildren integer DEFAULT 0,
   \ldots the actual node data
);
```

The field "nchildren" is a counter for the number of children that the node has *ever* had; we assume here it is not updated when nodes or subtrees are deleted.

Section 4 provides a complete implementation of these operations for fgID in PostgreSQL.

3.1 Computing the level of a node

Cost: *string operations (no table access)*

This is a pure string operation, no table access is required.

- **vgID:** count the number of separators ('/') in the PK
- **fgID:** count the number of characters in the PK, divide by the fixed size of the counters.

3.2 Selecting or deleting a subtree

Cost: *index scan of the tree*

- **vgID:** The subtree rooted at /26/5/7 is selected by

  ```
  ... WHERE id LIKE '/26/5/7%' AND id < '/26/5/70'
  ```

- **m-vgID:** The subtree rooted at /126/05/07 is selected by

  ```
  ... WHERE id LIKE '/126/06/07%'
  ```

- **fgID:** The subtree rooted at 260507 is selected by

  ```
  ... WHERE id LIKE '260507%'
  ```

3.3 Selecting the direct children of a node

Cost: *index scan of the tree*

- **vgID:** The direct children of /26/5/7 are selected by

  ```
  ... WHERE id LIKE '/26/5/7/%' AND id NOT LIKE '26/5/7/%/%'
  ```

- **m-vgID:** The direct children of /26/5/7 are selected by

  ```
  ... WHERE id LIKE '/126/06/07/%' AND id NOT LIKE '/126/05/07/%/%'
  ```

- **fgID:** The direct children of 260507 are selected by

  ```
  ... WHERE id LIKE '260507%' AND char_length(id) = (char_length('260507')+2)
  ```

3.4 Inserting a node or a subtree

Cost: *index scan of the tree + string and math operations*

Insertion is a procedural operation. As each RDBMS has a different way of defining procedures, we will just describe here the necessary steps. Examples for PostgreSQL are provided in 4.

In order to insert a new child of "daddy" (either one of /26/5/7, /126/05/07 or 260507 in the examples above) you have to

1. add one to the number of children of "daddy"

```
UPDATE tree SET nchildren = (nchildren + 1) WHERE ID = ''daddy'';
```

2. encode the number of children of "daddy" in base 160, bring it to the correct format depending on the variant of gID (pad with 0 or not, prepend a digit counter or not, prepend / or not, count down or up, ...) and append it to daddy's gID to obtain the new node's gID.

3. insert the new node

When inserting a subtree, the index of the root of the subtree has to be computed as above, and prepended to the index of each node of the subtree before insertion.

Remark that only the parent node has to be updated on insertion.

3.5 Selecting the ancestors of a node

Cost: *index scan of the tree*

You can specify all ancestors of a node in a single SQL statement; for instance for vgID

```
... WHERE '/25/6/7' LIKE (id || '/%') AND id < '/25/6/7'
```

The second part of the clause, while logically redundant, is a "hint" to the optimizer. At least in PostgreSQL, without it the optimizer will choose a sequential scan of the table and disregard the index.

3.6 Selecting all leaves

Cost: *scan of the tree*

A leaf is a node without descendants: it has 0 children. Hence

```
... WHERE nchildren = 0
```

If this type of query is often necessary, you may be well advised to keep an index on tree(nchildren).

3.7 Determining if A is a descendant of B

Cost: *string operations, no table access*

This is a pure string operation on the indices, no table access is necessary.

4 Putting it all together: a PostgreSQL implementation

http://www.postgresql.org/mhonarc/pgsql-sql/2000-04/msg00267.html

We describe here a small package that can be used for implementing gID on PostgreSQL. It can be found at <http://...>

The package uses the procedural language PL/PGsql. A better implementation would probably define the gID as new Postgres types, and code all this in C.

The files should be loaded in alphabetical order.

4.1 tree0_encoding.sql

This file defines and populates the table _b160_digits of "digits" in base 160,

```
CREATE TABLE \_b160\_digits (deci integer, code char);
```

and the two functions

```
CREATE FUNCTION \_b160\_encode(integer) RETURNS string
  AS '....' LANGUAGE 'plpgsql';
CREATE FUNCTION \_b160\_encode(integer,integer) RETURNS string
  AS '....' LANGUAGE 'plpgsql';
```

The first function returns a variable size encoding; the second a fixed size encoding (the second parameter is the size), and raises an exception if the number is too large to be represented with the requested number of digits.

4.2 tree1_define.sql

This file provides a function

```
CREATE FUNCTION _tree_create(text,integer,text,text) RETURNS bpchar
  AS '....' LANGUAGE 'plpgsql';
```

that creates a tree infrastructure of either fgID or vgID. Assuming you have a table "mytable" with primary key "myid", then calling

```
SELECT _tree_create('mytree',2,'mytable','myid');
```

will cause:

1. the creation of a table

   ```
   CREATE TABLE mytree_bkg(
       gid text PRIMARY KEY,
       nchildren int,
       sid integer REFERENCES mytable(myid)
   );
   CREATE UNIQUE INDEX mytree_bkg_sid ON mytree_bkg(sid);
   ```

   for the tree structure.

2. the creation of a view

   ```
   CREATE VIEW mytree AS
      SELECT t.gid,n.*
            FROM mytable n, mytree_bkg t
            WHERE t.sid=n.myid;
   ```

   with: a trigger on UPDATE that blocks updating the gid and allows updating the node data, a rule on DELETE that deletes the corresponding entry both in mytree_bkg and mytable, and a trigger ON INSERT that raises an exception and informs the user to use the insertion function described below.

3. two insertion functions that compute automatically the gID of the new node:

   - a function mytree_insert(text,text,integer,text) for insertion simultaneosly in both tables: mytree_insert('2201','hello',0,'not much') inserts a new child of 2201 with data1='hello', data2=0 and data3='not much'

- a function mytree_insert_node(text,integer) for insertion in mytree_bkg
  mytree_insert('2201',25) inserts in mytree_bkg a new child of 2201 with sid=25

4. a function mytree_move(text,text) that moves subtrees:
   mytree_move('2201','23') moves the subtree rooted at 2201 to a place below 23 (maybe 2307)

5. a function mytree_len() that returns the length of the encodings used in the gID (2 here; 0 if variable size).

5 Non-tree hierarchies

sequence as id, table with (id,g-index) with possibly many g-indices for each id (if TOO many, bad model: list all genealogies, i.e., paths from the root)

References

[1] Philip Greenspun, *Trees in Oracle SQL*, in **SQL for Web Nerds**
<http://photo.net/sql/trees.html>

[2] Joe Celko, *SQL for Smarties*, in **DBMS Online**, March to June 1996
<http://www.dbmsmag.com/9603d06.html>
<http://www.dbmsmag.com/9604d06.html>
<http://www.dbmsmag.com/9605d06.html>
<http://www.dbmsmag.com/9606d06.html>

[3] Graeme Birchall, **DB2 UDB V6.1 SQL Cookbook**,
<http://ourworld.compuserve.com/homepages/Graeme_Birchall/HTM_COOK.HTM>