

Tree Structured Data in a Relational Database

Jamie Hogle

<http://www.geniegate.com/>

Consultant, GenieGate

ABSTRACT

Perhaps nothing has revolutionized the information industry more than the relational database. However, a relational database isn't particularly logical for hierarchical data structures, such as categorized directories or catalogs. This document explores one strategy for dealing with this problem.

I am a freelance consultant, dealing with all things UNIX, perl, PHP, java and more. You might consider hiring me if you are in need of unix (Linux, FreeBSD, OpenBSD, etc..) services.

1. The Project

Creating a categorized index of internet radio programming, arranged in a tree like structure, similar to Yahoo or dmoz, such that program directors and stream owners might navigate the directory, procuring content for their radio station or internet stream. †

The directory featured a search function, however, this isn't useful when browsing new programs, after-all, which keywords would you use?

2. The Challenges

In any project, there are challenges. The system has to run on an older, already over-taxed piece of hardware servicing other tasks.

2.1. Choosing Perl

In addition to being user friendly and easy to navigate, it had to be **fast** and responsive or people would abandon the site.

For these reasons, I chose to implement the system using *perl* instead of PHP, Java was a close second, but the resource requirements were too excessive. PHP is good for simple web sites, but not suitable for serious web applications, perl allows us to run under FastCGI or plain vanilla CGI if need be.

This was a difficult decision, had PHP been usable, GenieGate might have been used for the user management.

2.2. The Database

Original drafted plans called for flat text files, after recognizing the other needs (such as transactions and referential integrity) these plans were scrapped in favor of a relational database.

Postgresql was used instead of mysql, again, difficult decision, mysql would have been easier, it was already installed and in use for other tasks, but I wanted robust performance with *good* integrity checks, as we shall see, this turned out to be a good decision.

† <http://streamsyndication.com/>

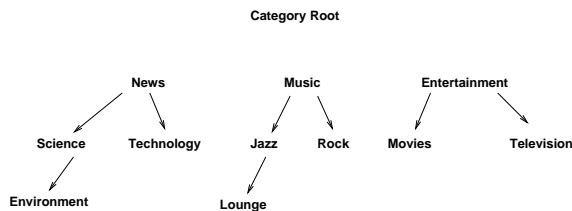
2.3. Lean and Simple

We like applications to be clean, simple and lean. There is a simple axiom to remember, *zero code, zero bugs, zero time.*

Your ultimate goal should therefore be, to use zero lines of code (this, of course, is impossible, but you get the idea)

3. Database Design

The data is organized in a *tree-like* structure.



Structures such as News/Science/Environment do not fit within the constraints of a relational database very well. We had a central table, called *directory* which contained one entry for each category as well as a *parent ID* representing the category above it.

Tree in a Relational Table

ID	Parent	Path	Description
1	(null)	Food	Food Stuff
2	1	Bread	Bread Stuff
3	2	Cake	Food/Bread/Cake

This figure represents the entries for Food/Bread/Cake where the Food entry is a root category and therefore has no parent ID, Bread has an ID of 2, with its parent as Food, while Cake has an ID of 3, with its parent ID of 2, meaning that Cake is a *member of* Bread.

The path column is not unique! we could have /Music/70s/Bread or /Wedding/Party/Cake, this non-unique nature of the immediate path has implications we will explore later.

We could obtain all the *immediate* food items by using this query:

```
SELECT * FROM directory WHERE Parent=1
```

This works for single entries, but fails on recursive queries, we *could* simply query for each entry searching out the parent ID, then repeating the process, however a path such as *Food/Bread/Cake/Chocolate/Donuts* would involve **five round-trip queries** to the database!

3.1. Postgresql to the rescue

To solve the round-trip issue, we used a stored procedure within postgresql to perform the above queries without multiple round trips to and from the database.

Our first function is internal, used within the loop of the second function, given a title and a parent ID, it returns the parent path.

3.1.1. Use stored procedures to avoid round-trips

```
CREATE FUNCTION internal_directory_find_directory_did(INTEGER, VARCHAR)
RETURNS VARCHAR AS $$
DECLARE
    path VARCHAR;
BEGIN
    IF $1 IS NULL THEN
        SELECT did INTO path FROM directory WHERE pid IS NULL AND title=$2;
    ELSE
        SELECT did INTO path FROM directory WHERE pid=$1 AND title=$2;
    END IF;
    RETURN path;
END;
$$ LANGUAGE plpgsql;
```

Our second, "public" function, accepts a path in the form of Food/Bread/Cake and parses it out, looping until it reaches the end. Note that simply searching for "Cake" wouldn't work, as other top level categories could potentially have the same name, such as Wedding/Cake.

```
CREATE FUNCTION lookup_directory_path(VARCHAR) RETURNS INTEGER AS $$
DECLARE
    did INTEGER;
    ctr INTEGER DEFAULT 1;
    path VARCHAR;
    ipath VARCHAR;
BEGIN
    ipath = ltrim(rtrim($1, '/'), '/');
    LOOP
        path = split_part(ipath, '/', ctr);
        IF length(path) = 0 THEN
            EXIT;
        END IF;
        did = internal_directory_find_directory_did(did, path);
        ctr = ctr + 1;
        IF did IS NULL THEN
            EXIT;
        END IF;
        IF ctr > 1024 THEN
            RAISE EXCEPTION 'Exceeded path count';
            EXIT;
        END IF;
    END LOOP;
    RETURN did;
END;
$$ LANGUAGE plpgsql;
```

The above functions allow us to save the round trips to the database, when we need to know the ID for a given path.

All this is well and good, but what if we need to know the path for a given ID? as you have probably guessed, this too involves a stored procedure.

3.1.2. Lookup the path from an ID

```
CREATE FUNCTION lookup_directory_ancestor_path(INTEGER) RETURNS VARCHAR AS $$
DECLARE
    id INTEGER;
    title VARCHAR;
    path VARCHAR DEFAULT '';
    lq directory%ROWTYPE;
BEGIN
    id=$1;
    LOOP
        SELECT * INTO lq FROM directory WHERE did=id;
        IF lq.title IS NULL THEN
            EXIT;
        END IF;
        title=lq.title;
        id=lq.pid;
        path = title || '/' || path;
    END LOOP;
    path=ltrim(rtrim(path,'/'),'');
    RETURN path;
END;
$$ LANGUAGE plpgsql;
```

The above function, given an ID, allows us to lookup the pathname in a manner that is intuitive, without expensive network queries to and from the database.

The above function makes the following SQL statement possible.

```
SELECT * FROM directory WHERE lookup_directory_ancestor_path(id)
LIKE 'Food/%';
```

Now we have an intuitive way of traveling through the category index using SQL commands, treating the path names as strings. With an index, the above function could actually avoid the whole traversal.

3.2. Pathname Integrity Checking

Earlier on, I said I wanted good data checking, too often this kind of checking is done on the application level rather than the database level. The problem with this approach is that validation becomes some-what relaxed and it's easy to have corrupted tables.

Besides the mundane checks, such as ensuring the title has no tabs or newline characters, foreign key constraints (and in this case, key constraints on the parent ID) we also want to ensure the complete pathname is unique.

This is allowed `Music/70s/Bread` and `Food/Bread`. This is forbidden `Food/Bread` and `Food/Bread` (just as you can't have identical directory names on a filesystem, you can't have two identical categories in our table)

Once again, postgresql comes to the rescue with constraints.

First, we need to write a simple function that verifies no such title exists *within the same level*, simply checking the title for uniqueness is insufficient, recalling our Bread example.

```
CREATE FUNCTION is_unique_directory_title(INTEGER,INTEGER,VARCHAR)
RETURNS boolean AS $$
DECLARE
    flag boolean;
BEGIN
    IF $2 IS NULL THEN
        SELECT '1' INTO flag FROM directory WHERE did !=$1 AND pid IS NULL AND title = $3;
    ELSE
        SELECT '1' INTO flag FROM directory WHERE did !=$1 AND pid = $2 AND title = $3;
    END IF;
    IF flag THEN
        RETURN '0';
    ELSE
        RETURN '1';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Then, use our function to check to the table for integrity, did is the directory ID, while pid is the *parent* id.

```
ALTER TABLE directory ADD CHECK(is_unique_directory_title(did,pid,title));
```

Now Postgresql will refuse to accept entries that have identical titles on the same level.

4. Lessons Learned

Relational databases are wonderful, but some data structures don't work particularly well. Stored procedures are complex and should generally be avoided, but when used judiciously, they can save you the overhead of repeat queries. (just don't over-do them!)

Postgresql offers the ability to use other languages, such as perl, but this seemed like overkill. The jury isn't out yet on whether or not perl should have been used within postgresql.

Using perl along with functions that access the filesystem (such as a custom format specifically designed for tree structures) require elevated privileges.