

1 Dealing with Hierarchies

There are many real world situations where the relational model doesn't fit naturally. One of them are generalizations and specializations from object oriented languages. Others involve recursive data structures, such as as trees.

2 Generalization/Specialization

The terms 'generalization' and 'specialization' are fancy names to refer to the parent and child classes. For example, if we define (in Java):

```
public class Square extends Shape {  
    // some code.  
}
```

We've defined a 'Square' class, with all of the properties of a 'Shape'. Every square is a shape (that's generalization), but not every shape is a square (that's specialization).

The first task we're generally concerned with is how to store the data in the database. For this, we have several options: store data in multiple tables, store data in one wide table, or some hybrid of the two.

2.1 Multiple Tables

Storing data in multiple tables will involve creating two tables. The shape table is the 'parent':

```
create table SHAPE (  
    shapeid      int ,  
    typ          int ,      — if 1, then square, etc.  
    description  varchar(10),  
);
```

The square table is a bit tricky. It will also have the shapeid:

```
create table SQUARE (  
    shapeid      int ,  
    side         int      — size of the side.  
);
```

Everytime we grab object data that extends Shape, we need to first select data from SHAPE, then depending on the 'typ' field, pick the right table, and select the 'extra' data from the appropriate table using the primary key from the parent table.

Notice that in this scheme we can easily have sub-sub classes. The only practical concern is that grabbing these things causes an extra database hit, something you should always avoid.

2.2 One Tables

Another approach is to store everything in one large table, with possibly many nullable fields. For example:

```

create table SHAPE (
  shapeid      int ,
  typ          int ,      — if 1, then square, etc.
  description  varchar(10),
  side        int ,      — square property
  radius      int ,      — circle property
  — etc
);

```

Now, if ‘typ’ indicates a square, the radius attribute will be null. If ‘typ’ indicates a circle, then radius will be set, while side will be null. This is something that will need to be handled at the application level—as databases aren’t usually flexible enough to enforce things at such a level.

3 Trees

Storing trees and other hierarchical data presents an extra complication. In procedural languages, we setup a node based data structure, ie:

```

// java binary search tree node
class Node {
  public Comparable data;
  public Node left ,right;
  public Node(Comparable d,Node l,Node r){
    data = d;
    left = l;
    right = r;
  }
}

```

We can then write code to manipulate these notes relatively easily, ie:

```

// add stuff into a binary search tree.
public Node insert(Node t,Comparable i){
  if(t == null)
    return new Node(i, null, null);
  if(i.compareTo(t.data) <= 0){ // insert into left
    t.left = insertTree(t.left, i);
  }else{ // insert into right
    t.right = insertTree(t.right, i);
  }
  return t;
}

```

One may insert a bunch of items into such a tree via:

```

Node t = null;
for(int i=0;i<10;i++){
  t = insert(t,new Integer((int)(Math.random()*100));
}

```

Now, this all seems very easy, but how do we do similar things in a database? There are primarily two methods that stand out above the rest: one is to use some sort of linking between “nodes” (just like in the Java example above), and another method is to encode the path from root to the node in some attribute, usually via numerical values or via some ‘path’.

3.1 Adjacency Model

We start by defining our “node” data container. For example, dealing with the standard ‘manager’ example, we can create a simple employee table:

```
create table EMPLOYEE (  
    employeed int ,  
    name varchar(20) ,  
    managerid int ,  
    — etc  
);
```

We can obviously have other fields in there (and define indexes, etc), but the point is to have some link back to the table itself. Note that unlike in Java, this relationship works both ways. For any manager, we can easily get all employees, and for any employee we can easily find the manager.

3.1.1 Non-Traversal Traversal

Often whenever you’re faced with such data structures, you don’t *really* need to loop through all nodes. For example, imagine this is a web application to display employee information. We have a page to display employee information, and a link to the manager. It takes 1 database hit to render that page. Similarly, when the user clicks on the manager link, it’s also trivial (1 database hit) to grab manager information.

This can often be applied to online file repositories. For example, you can manage a directory structure in a database using such links, and because you don’t ever traverse the whole tree in any one query, there are no issues (users just clicks on sub-directories, or the parent directory).

3.1.2 The Traversal: connect by

There are several ways of traversing this data. If you only need a few layers (lets say this is a forum, and you only wish to display the top 2 layers), then you can do an outer self join. For example, to grab manager and all of the employee information for that manager, you might do:

```
select ...  
from EMPLOYEE a outer join EMPLOYEE b  
    on a.managerid=b.employeed
```

This way, you’ll get just 2 layers of this ‘tree’. Once in a while, this may not be enough, and you’ll need to resort to the actual traversal.

The bad news is there's nothing in the standard SQL to allow you to do the hierarchy traversal—so whatever method we come up with will be database dependent.

One obvious solution is to use PL/SQL (or T-SQL) to traverse the tree. This is generally and should be avoided (in fact, avoid procedural things whenever coding database code).

Another solution is presented by Oracle's 'connect on' feature. For example, we can find out all the employees who work under a certain manager (managerid=42), simply by doing:

```
select employeeid ,name
from EMPLOYEE
connect by managerid = prior employeeid
start with managerid=42
```

So if there's 8 layers of managers between you and managerid=42, you'd show up in the query. Note that you can reverse the 'connect by' clause, to lookup all managers for a particular employee, etc. For more info, google for it!

3.1.3 The Traversal: connect by Sample

We start by defining our 'node' table:

```
drop table mynode;
create table mynode (
    nodeid int ,
    parentid int ,
    name varchar(100)
);
```

We then create a short Perl script to generate data for our table (you can do this by hand, or let the computer do it).

```
use strict;

# insert value into a binary search tree (keep track of parent).
sub insert {
    my ($t,$o,$p) = @_;
    return {o=>$o,p=>$p} unless $t;
    if($o <= $t->{o}){
        $t->{left} = insert($t->{left},$o,$t);
    }else{
        $t->{right} = insert($t->{right},$o,$t);
    }
    return $t;
}

# populate the tree (balanced binary)
my $t;
for (50,25,13,30,7,15,28,35,75,65,85,60,67,80,90){
    $t = insert($t,$_);
}
```

```

# output the tree
my @s = ($t);
while(@s){
    my $s = shift @s;
    printf("insert into mynode values(%d,%d, 's%d:%d '); \n",
        $s->{o}, $s->{p} ? $s->{p}{o} : 0,
        $s->{o}, $s->{p} ? $s->{p}{o} : 0);
    push @s, $s->{left} if $s->{left};
    push @s, $s->{right} if $s->{right};
}

```

The output of the program will look something like this:

```

insert into mynode values(50,0, 's50:0 ');
insert into mynode values(25,50, 's25:50 ');
insert into mynode values(75,50, 's75:50 ');
insert into mynode values(13,25, 's13:25 ');
insert into mynode values(30,25, 's30:25 ');
insert into mynode values(65,75, 's65:75 ');
insert into mynode values(85,75, 's85:75 ');
insert into mynode values(7,13, 's7:13 ');
insert into mynode values(15,13, 's15:13 ');
insert into mynode values(28,30, 's28:30 ');
insert into mynode values(35,30, 's35:30 ');
insert into mynode values(60,65, 's60:65 ');
insert into mynode values(67,65, 's67:65 ');
insert into mynode values(80,85, 's80:85 ');
insert into mynode values(90,85, 's90:85 ');

```

We can now write a sql query to do a breadth first traversal of the tree, starting with node 25 (left child of root):

```

SQL> select nodeid,parentid,level,name
       2 from mynode
       3 connect by prior nodeid = parentid
       4 start with nodeid=25
       5 order by level,nodeid;

```

NODEID	PARENTID	LEVEL	NAME
25	50	1	s25:50
13	25	2	s13:25
30	25	2	s30:25
7	13	3	s7:13
15	13	3	s15:13
28	30	3	s28:30
35	30	3	s35:30

7 rows selected.

3.1.4 The Traversal: with

There's also the "with" feature, which appears in other databases. Here's how this works:

```
with temptable (employeeid ,name) as (  
  — base case, get all employees whose manager is managerid=42  
  select employeeid ,name  
  from employee  
  where managerid=42  
  union all  
  — union all with the recursive case  
  select employeeid ,name  
  from employee , temptable  
  where employee.managerid = temptable.employeeid  
)  
select employeeid ,name from temptable ;
```

In essence, we setup a recursive table. We need to specify the base case (where do we start), and union that with the step case, how do we get to the next record (join with original table).

3.2 Materialized Path Model

With the materialized path model, we store the path from root node upto the 'node' we are dealing with. For example:

```
create table employee (  
  employeeid      int ,  
  name           varchar (20) ,  
  path           varchar (100)  
);
```

The path for the top most manager may be something like "1". The 2nd layer of managers will have paths such as: "1.1", "1.2", "1.3", etc. The children of "1.3" node will have paths like "1.3.1", "1.3.2", "1.3.3", etc. This string just keeps on growing.

This approach is very flexible and doesn't require a lot of processing time—which is why many forums that support nested comments implement this.

To get all employees under a certain manager, simply do:

```
select e.employeeid , e.name  
from employee m, employee e  
where m.path like e.path || '%';  
and m.employeeid=42
```

The reason the above works is because employees (or rather child nodes) will always have a longer 'path' string than the parents.

4 XML Databases

For hierarchical data, using XML to store data may be the most natural approach. Parsing XML is surprisingly easy. The below Perl module, `pxml.pm`, parses and queries XML data:

```
#####
## Particle's simple XML Module, Version 1.1
## (not designed for speed)
## (C) 2003-2008, Alex S. <alex@theparticle.com>
## http://www.theparticle.com/
#####

package pxml;

use strict;

#####
## create a new instance of xml object.
#####
sub new {
    my ($class, %args) = @_;
    my $this = { %args };
    if ($args{file}){
        my $root = parse(loadFile($this->{file}));
        $this->{$_-} = $root->{$_-} for keys %{$root};
    }
    return bless $this, $class;
}

#####
# xml parser; returns root node
#####
sub parse {
    local $_ = shift;
    my @strings;
    s{<![CDATA\[([.*?])\]]>}{push @strings,$1;'<#'. $#strings.'/>'}xsge;
    s|<!--.*?-->||gs;    s|<\/?.*?\?>||gs;
    push my @stack, { type=>'root', value=>'~', ch=>[] };
    while(m|(.*)<(.*?)>|gs){
        for(my $t = $1){
            s/^\s+|\s+$//gs; s/\s+//gs; s|&lt;|<|gs; s|&gt;|>|gs;
            s|&amp;|&|gs; s|&apos;|'|gs; s|&quot;|\"|gs; s|#\"'
            push @{$stack[-1]->{ch}},
                new pxml(type=>'text',value=>$_) if length;
        }
    }
    my $t = $2;
    if($t !~ m|/|){ # opening tag
        my ($tag,$attr) = split /\s+/, $t, 2;
```

```

    push @stack,
        new pxml(type=>'tag', value=>$tag, attrib=>{}, ch=>[]);
    $stack[-1]->{attrib}{$1} = $2
        while($attr =~ m|\s*(.*?)="(.*?)"|g);
} elsif($t =~ s|^/|){ # closing tag
    my $s = pop @stack;
    push @{ $stack[-1]->{ch} },$s if $t eq $s->{value};
} elsif($t =~ m|#[\d+]|){ # CDATA section
    push @{ $stack[-1]->{ch} },
        new pxml(type=>'text',value=>$strings[$1])
        if $strings[$1];
} elsif($t =~ s|/$|){
    my ($tag,$attr) = split /\s+/, $t, 2;
    my $o =
        new pxml(type=>'tag',value=>$tag, attrib=>{}, ch=>[]);
    $o->{attrib}{$1} = $2 while($attr =~ m|\s*(.*?)="(.*?)"|g);
    push @{ $stack[-1]->{ch} },$o;
}
}
return $stack[0]{ch}[0];
}

```

```

#####
# query code
#####
sub query {
    my ($t,$q) = @_;
    my @stack = ([ '/' . $t->{value}, $t, $t->{type} ]);
    my @matched;
    while(@stack){
        my ($name,$val,$type) = @{pop @stack};
        if($name =~ m/Q$q\E$/i){
            push @matched,$val;
        } elsif($type eq 'tag'){
            push @stack,
                [$name.'/' . $_->{value}, $-, 'tag', ]
                for grep { $_->{type} eq 'tag' }
                reverse @{ $val->{ch} };
            push @stack,
                [$name.'/' . $_. '$', $val->{attrib}{$_}, , ]
                for sort keys %{$val->{attrib}};
            my $text = join(' ',map { $_->{value} }
                grep { $_->{type} eq 'text' }
                reverse @{ $val->{ch} });
            push @stack,[$name.'$', $text, , ] if $text;
        }
    }
}

```

```

    return wantarray ? @matched : $matched[0];
}

#####
# load file
#####
sub loadFile {
    local $_ = shift;
    open my $in, $_ or die $!;
    local $/=undef;
    return <$in>;
}

1;

```

So given an XML file such as:

```

<meta>
  <!-- tables -->
  <table name="user">
    <col name="id" type="number" precision="10" auto="Y" pkey="Y" />
    <col name="username" type="varchar" length="64" required="Y" />
    <col name="password" type="varchar" length="64" required="Y" password="Y" />
    <col name="name" type="varchar" length="64" required="Y" />
    <col name="email" type="varchar" length="64" required="Y" />
    <uniq name="username"/>
  </table>
  <!-- .. -->
</meta>

```

The below code can be used to read (query, etc.) such files:

```

#!/usr/bin/perl
# xml parser test script.
use strict;
use pxml;

my $meta = new pxml(file=>'metadata.xml');

# loop for all tables.
for my $table ($meta->query('/table')){
    # find name of table.
    my $name = $table->query('/name$');
    print "drop_table_$name;\n\n";
    print "create_table_$name_\n";

    # loop for all columns in this table.
    print join ",\n", map {
        # get column information
        my $name = $_->query('/name$');
    }

```

```

    my $type = $_->query('/type$');
    my $length = $_->query('/length$');
    my $auto = 'AUTO_INCREMENT' if $_->query('/auto$') eq 'Y';
    "____$name_$type_" . ($length ? "($length)" : ""). "_$auto"
} $table->query('/col');

# get name of all columns marked as primary key.
my $pkey = join(',',map { $_->query('/name$') }
    grep { $_->query('/pkey$') eq 'Y' } $table->query('/col'));
# output joined primary key (ie: k1,k2,k3, etc.)
print ",\n____primary_key($pkey)" if $pkey;
print "\n);\n\n";
}

```

The output is about what you'd expect (thought no very polished):

```
drop table user;
```

```
create table user (
  id number AUTO_INCREMENT,
  username varchar (64) ,
  password varchar (64) ,
  name varchar (64) ,
  email varchar (64) ,
  primary key(id)
);
```

This is something that can obviously be expanded upon.

5 Conclusion

Hierarchical data is a pain in the neck when you have to deal with it in a database.