Looping the Loop: Different Ways of Working with Recursive Structures

Dr. Paul Dorsey
Michael Rosenblum
Dulcian, Inc.
www.dulcian.com

NYOUG - June 7, 2011

Overview

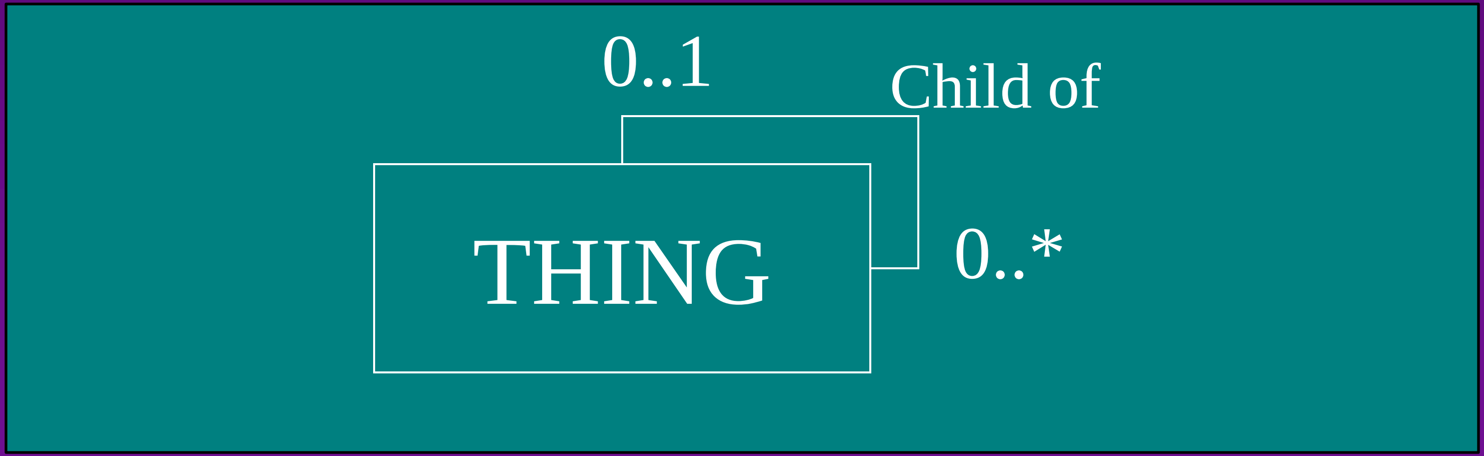
- Recursion
  - Powerful modeling technique
  - Can be used for a number of reasons
    - Linked lists (contract versions)
    - Storage of tree structures (organizational hierarchy)
  - Makes PL/SQL code more efficient
- Issues
  - Why is recursion underutilized?
  - Is there anything new?

Pre-Requisites

- This presentation assumes that:
  - 1. You know what a basic recursive table looks like.
  - 2. You have used the CONNECT BY clause.
  - 3. You can correctly place PRIOR in your code (at least on the second try ☺).

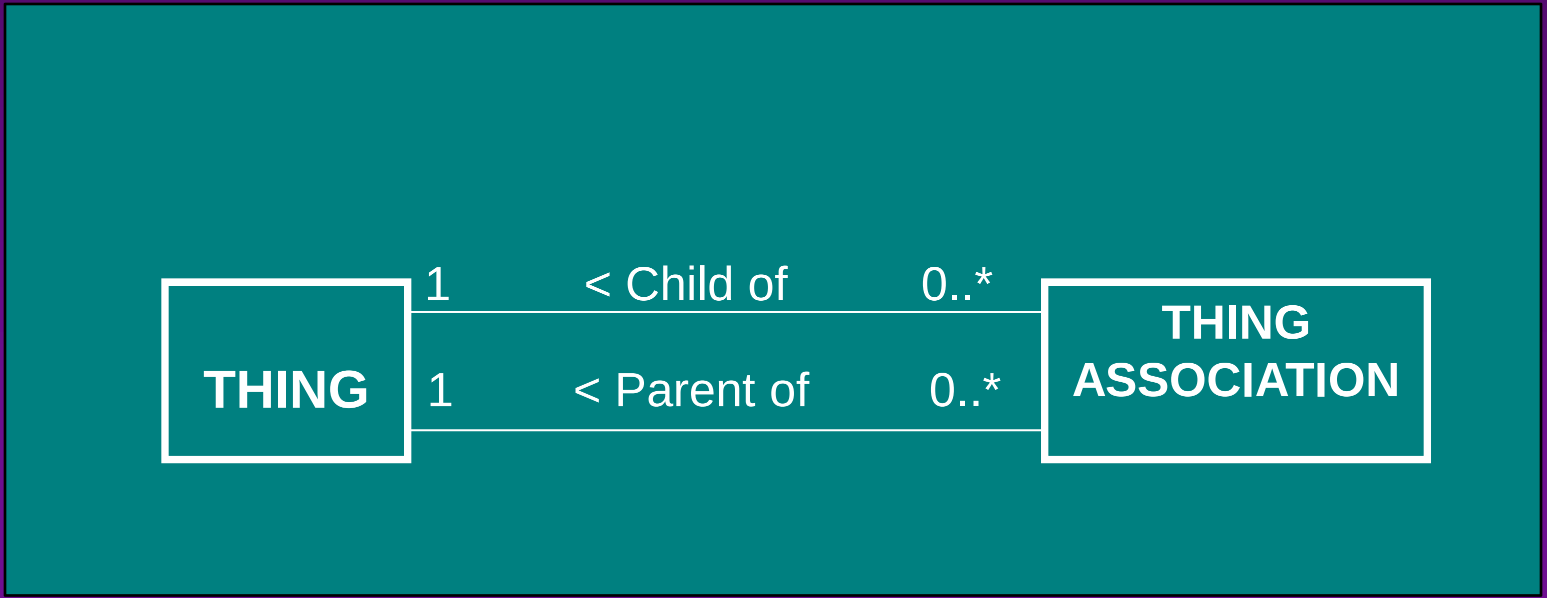
Simple Recursion

- Not as useful as one might think:
  - Data values tend to change over time.
  - Those changes are “of interest” (meaning they must be kept).



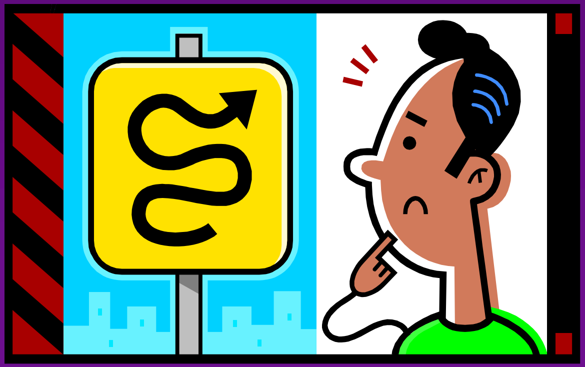
Pseudo-Recursion Model

- Alternative models to support "versions" – BAD IDEA
  - Add Start Dates and End Dates everywhere.
  - Place VERSIONING table off to the side of the THING table.
  - Create generic THING/THING ASSOCIATION structure.

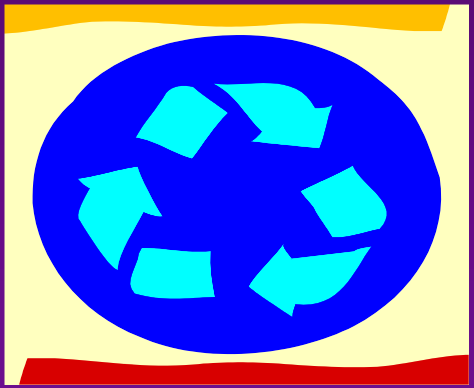


Hmm…

- People are building incomprehensible models.
- Simple recursion is inadequate to model business needs.
- Is there an alternative?
  - YES!



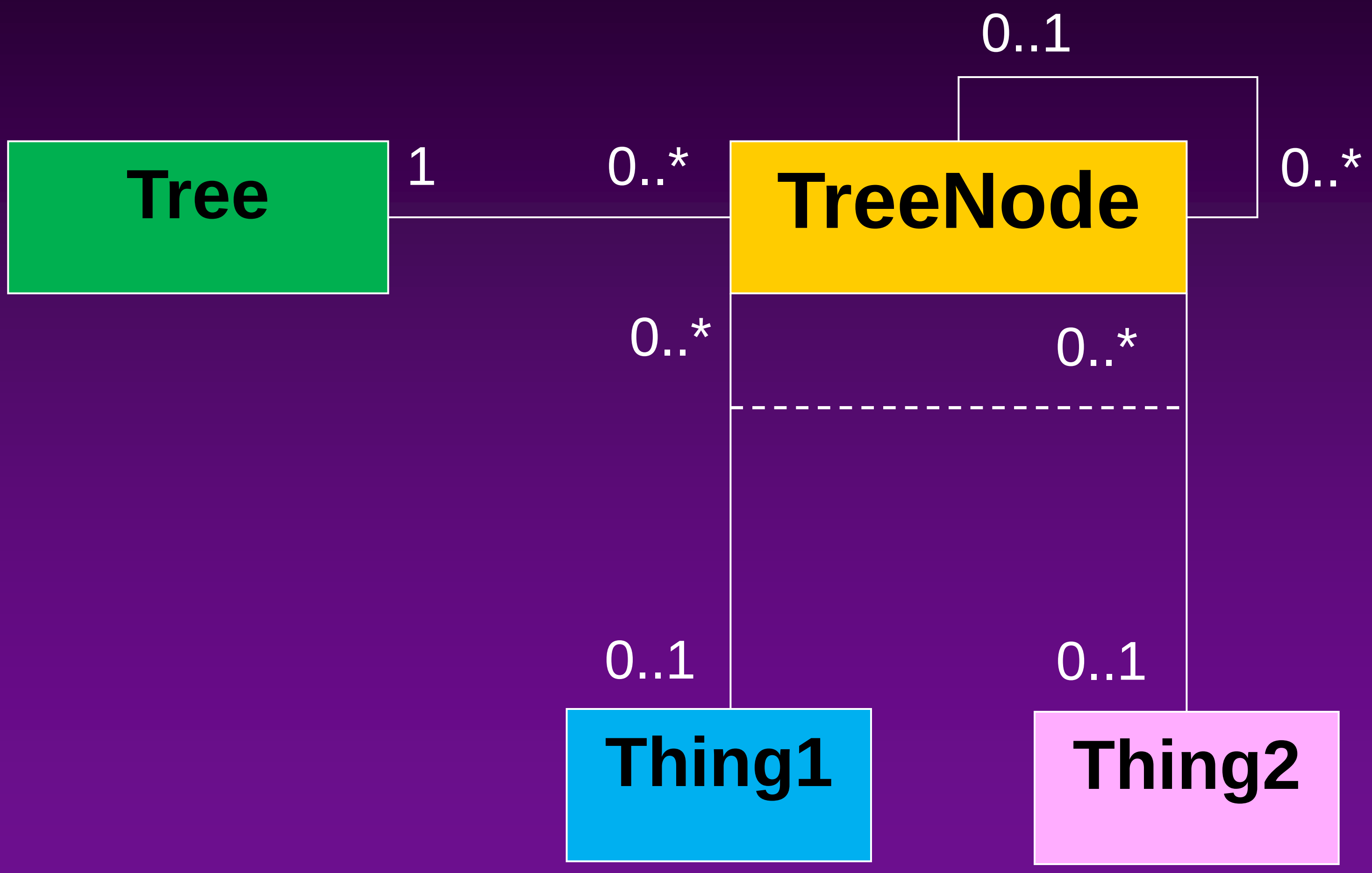
Using Recursion



Business Case

- <u>System Requirements:</u>
  - Organizational tree structure with 6 levels
    - The tree changes over time.
    - People are also parts of the tree (not just organizations)
    - Scheduling future changes
  - Reporting
    - Historical reports should use all trees between Start Date and End Date.
    - Events roll up using the tree valid at the time of the event.

Basic Model

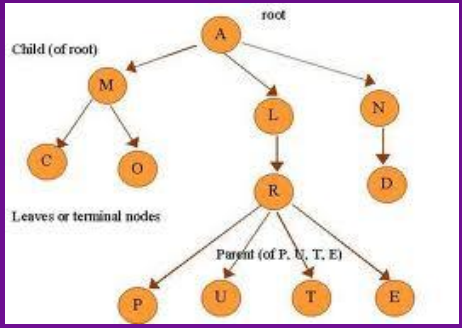


Tree Class

- Each object is a tree structure of a particular type for a period of time.
- Attributes:
  - Name: Logical name for the tree – rarely used
  - Description: Also a cool idea that is rarely ever used.
  - StartDate and EndDate: Dates for which the tree is valid.
  - Type: VERY important - type of the tree. In this example, “FUNCTIONAL” or “GEOGRAPHIC”.
    - Always include a Type attribute!
  - Status: Current, Future, Past, Potential

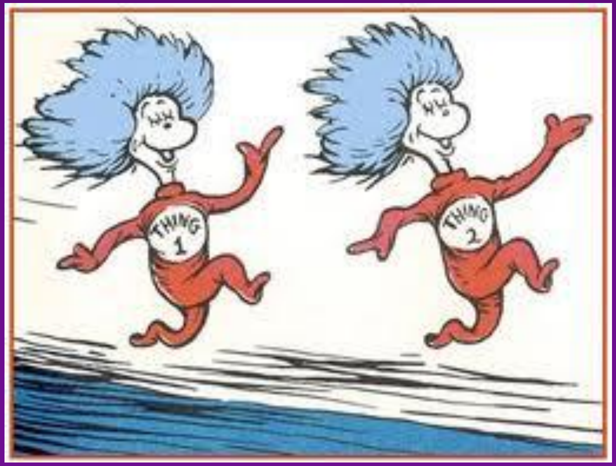
Tree Node Class

- Each object is a node.
  - Primarily a set of pointers to "Thing" classes.
  - Can also be a simple Folder node that does not point to any "thing" object.
- No Start/End Date attribute in this class.
  - Time dependency is only at the tree level.
- Attribute:
  - FolderName (populated only for a grouping folder)



Thing 1 / Thing 2 Classes

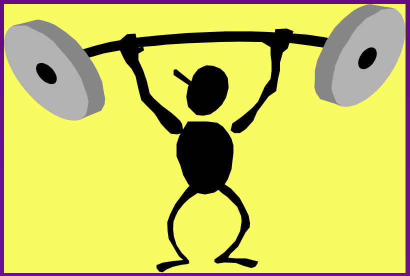
- Represent the standard object classes in your model
  - Could be OrgUnit or Person classes
- Demonstrate that you can have a tree with more than one kind of thing in it



Model Pros and Cons

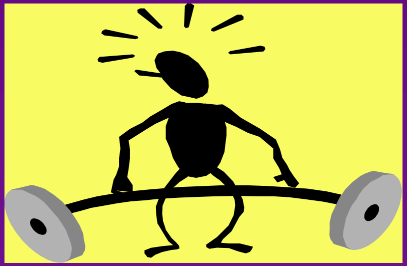
Model Strengths

- Each tree exists as its own recursive structure.
- Query is a simple recursive query.
- No need to deal with dates in node elements.
- Model pattern is reusable any time.
- Clear depiction of what is being modeled



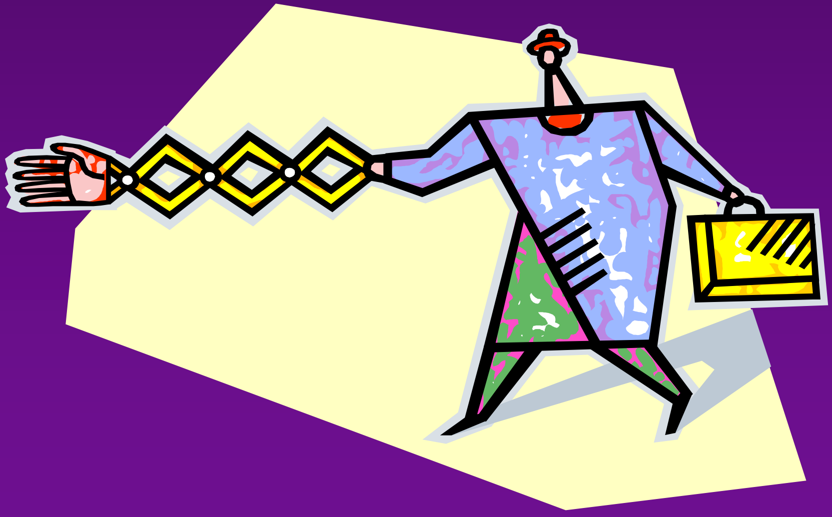
Model Limitations

- This model does not completely enforce everything.
  - Only one tree of a specific type can be valid for any date range.
  - Only specific types of Orgs are allowed to be children of other types of Orgs.



Model Extensions

- Questions:
  - How can you manage scheduled changes to the model (ones that did not yet happen)?
  - How can you correctly report over the desired period of time?



Implementing Future Trees

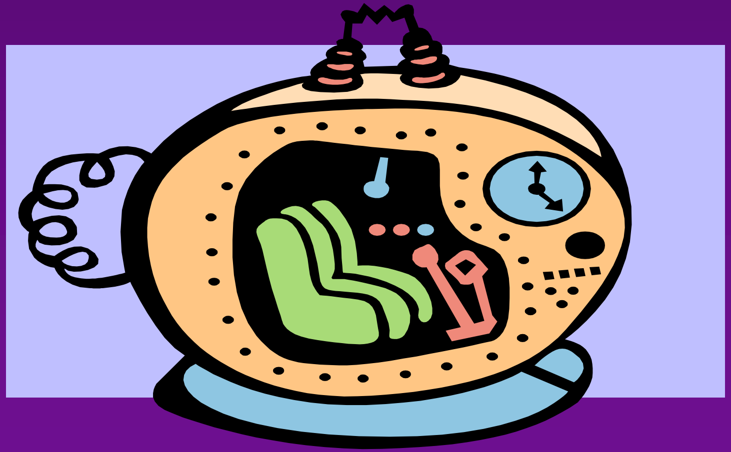
- Create special LOG table
  - Include complete description of the required change
  - Define moment when the change should be applied
  - Create AUDIT table to stores applied changes.
- Keep LOG table consistent
  - Mutually exclusive future changes null themselves out.
  - Meaningless future changes are automatically detected and removed.
- Apply changes using a database job, fired after midnight.
- Move successfully applied changes LOG → AUDIT
- Unsuccessful changes raise alarms

Maintaining Future Changes

- Create special temporary “future tree” as clone of current one.
  - User must enter requested date (“Tree date”)
  - Apply all previously scheduled changes to the “future tree”
- All changes are converted into an event in the LOG table
  - Scheduled date = “Tree date”
- When editing the tree, the temporary tree is removed
- Concurrent future modifications:
  - “Clone to the future” only part of the tree starting with the selected node
  - This root note (and all dependent nodes) must be locked until the future tree is removed.

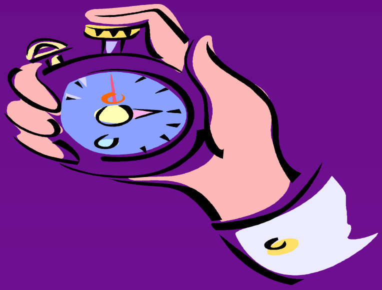
Time machine overview

- Reasons to do it:
  - Clear visibility of all changes and sequence of their application
  - Resolve all scheduling conflicts directly rather than using some type of complex analysis.



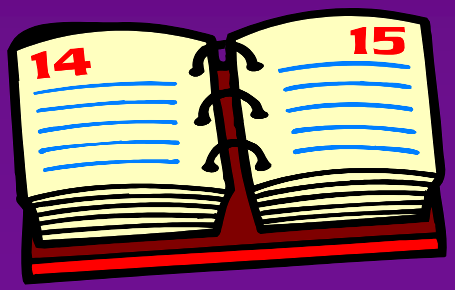
Handling Data "as of now"

- Problem is purely performance-related.
  - TreeDetail eventually grows to millions of rows.
  - Requests become expensive.
  - Usually there are a lot of requests about current information.
- Solution: "Current snapshot"
  - De-normalized MView (each levels = separate column)
    - Also include most often called data elements
  - Refresh is done either by request or during the midnight database job.
  - A lot of indexes!

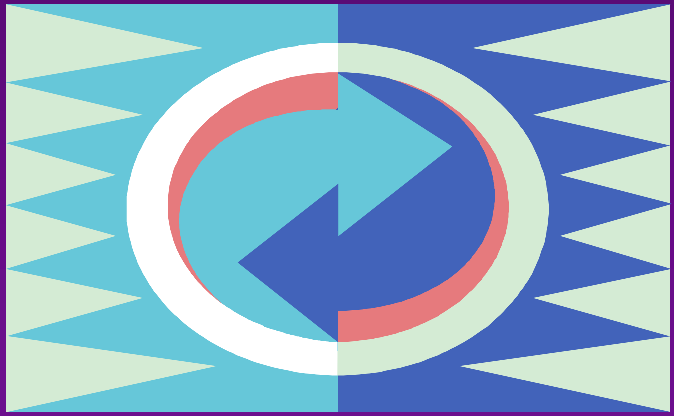


Handling Data Between Dates

- Reporting problem:
  - There may be many valid hierarchies over a period of time.
- Concept:
  - Log a <u>full hierarchical rollup</u> as of the moment of occurrence and exact timestamp for every critical event.
  - <u>Quickly</u> query which hierarchical chains were active.
- Solution to querying:
  - Appendable de-normalized table (each level = one column)
  - Shows the length of time that the <u>specified hierarchical chain</u> existed
  - Appended any time that a scheduled change to the organizational tree is being applied



Recursion in PL/SQL



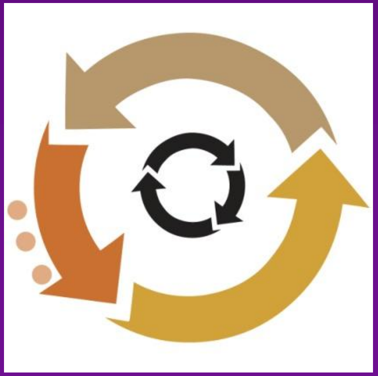
Recursion in PL/SQL

- Textbook example:

```
CREATE OR REPLACE FUNCTION f_factorial_nr
  (in_nr INTEGER) RETURN NUMBER AS
BEGIN
  IF in_nr in (0,1) THEN
    RETURN 1;
  ELSIF in_nr < 0 then
    RETURN NULL;
  ELSE
    RETURN(in_nr * f_factorial_nr(in_nr-1));
  END IF;
END;
```

Using Recursion

- Wrong case!
  - PL/SQL is rarely used for heavy mathematical tasks.
- Correct case should be:
  - Repetitive data-related processes
  - Always associated with recursive data structure
  - Completely different set of issues
    - Cursors
    - Variables
    - Transaction control
    - Exception handling



Cursors in Recursion

- Recursion in FOR-loops is a VERY BAD idea
  - All cursors are kept open until the end of the tree.
  - Concurrent users create scores of cursors.
  - Keeping multiple versions of data is very resource intensive.
- Right idea:
  - Bulk fetch into to the collection on each level
  - Spin through the collection

Bad Idea

```
function f_LevelDown_tx (i_fk number) return varchar2 is
  v_out_tx varchar2(32000);
begin
  for c in (select * from emp where mgr = i_fk) loop
    dbms_output.put_line(c.ename);
    v_out_tx:=f_LevelDown_tx(c.empno);
    if v_out_tx!='OK' then
       raise_application_error(-20999,v_out_tx);
    end if;
  end loop;
  return 'OK';
exception
    when others then
      return 'E:FK'||i_fk||'> error:'||sqlerrm;
end;
```

Good Idea

```
function f_LevelDown_tx (i_fk number) return varchar2 is
  v_out_tx varchar2(32000);
  type rec_tt is table of emp%rowtype;
  v_tt rec_tt;
begin
  select * bulk COLLECT into v_tt
   from emp where mgr = i_fk;
  if v_tt.count()>0 then
    for i in v_tt.first..v_tt.last loop
      v_out_tx:=f_LevelDown_tx(v_tt(i).empno);
      ...
    end loop;
  end if;
  return 'OK';
  ...
end;
```

Variables

- What is going on?
  - All local variables exist only in the current scope.
  - Two options to make variables visible:
    - 1. Passed down to the child call as input parameters
    - 2. Stored as global PL/SQL variables in a separate package

- Rule of thumb:
  - If the value is used directly in the child and nowhere else, it is a parameter
  - if the same value could be used in multiple places, it is a global variable
    - Scalar type if the value can be overridden
    - Collection type if multiple copies of the variable should be kept active

Example of Global Variables

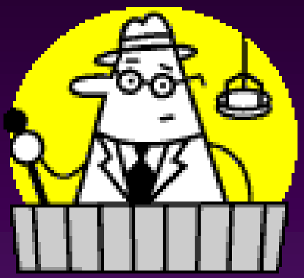
```
 ...
for i in v_tt.first..v_tt.last  loop
   if main_pkg.v_process_tt(v_tt(i).deptno)!=
     'Processing failed!'
   then
     v_out_tx:=f_LevelDown_tx(v_tt(i).empno);
   end if;

   if v_out_tx!='OK'  then
      main_pkg.v_process_tt(v_tt(i).deptno):=
      'Processing failed!';
   end if;
end loop;
...
```

Transaction Control

- What is going on?
  - If procedure is marked “autonomous transaction”
    - each recursive call would also spawn another autonomous transaction.
    - all transaction-level resources would be separate for each call
- Rule of thumb:
  - Try to avoid recursive autonomous transactions – may be too resource intense

Exception Handling



- Two main issues:
  - How to know precisely where an error occurred
  - What to do after the problem is detected.
- Error logging
  - Should be handled manually
  - Add used-defined variables (such as chain of input parameters) to Oracle's error stack
- Error handling:
  - Completely roll back changes to the moment before the recursive call.
  - <u>Good:</u> much simpler and significantly less complex than a full cleanup
  - <u>Bad:</u> May not be available in the case of explicit commits or autonomous transactions as part of a recursion.

Exception Handling Example

```
-- Main caller
declare
    v_tx varchar2(32000);
begin
    savepoint beforeLoop;
    begin
        v_tx:=f_LevelDown_tx(7839);
        -- business logic failure
        if v_tx!='OK' then
            rollback to savepoint beforeLoop;
        end if;
    exception
        when others then
            -- abnormal failure
            rollback to savepoint beforeLoop;
            raise;
    end;
end;
```

CONNECT-BY

CONNECT BY

- Oracle's method of working with recursive data:

```
select SYS_CONNECT_BY_PATH(empno,'|') path_tx,
       CONNECT_BY_ROOT ename root_tx,
       CONNECT_BY_ISCYCLE isCycle_yn,
       CONNECT_BY_ISLEAF isLeaf_yn,
       LEVEL level_nr,
       a.*
from emp a
start with mgr is null
connect by  nocycle mgr = prior empno
order siblings by ename
```

- Key elements:
  - Clause to link the parent/child structure
  - List of Oracle built-in functions

WHERE Clause

- Beware of performance trap with WHERE-clauses in the recursive query!

- Business case:
  - Get all tree nodes belonging to a certain tree
  - Data volume:1.6 million nodes, 800 trees (2000 nodes per tree) up to 6 levels deep
  - Indexes are created on all related columns (TreeNode PK, TreeNode RFK, TreeNode FK).

Handling WHERE Clause

Simplest Version (bad)

```
select *
from TreeNode
where Tree_oid = :1
connect by TreeNode_rfk =
  prior TreeNode_oid
start with
  TreeNode_rfk is null
```

- Takes 2 minutes to run
  - Reason: WHERE clause is applied only AFTER the whole lookup is finished → all trees are processed

Better Version

```
select *
from TreeNode
  where Tree_oid = :1
connect by TreeNode_rfk=
  prior TreeNode_oid
start with
  TreeNode_rfk is null
  and Tree_oid=:1
```

- Much better - 0.08 seconds
  - Reason: Only single tree is processed
- WHERE clause is redundant.

Best Solution

```
select *
from TreeNode
connect by TreeNode_rfk = prior TreeNode_oid
start with TreeNode_rfk is null and Tree_oid = :1
```

- No extra steps
- Simple way of finding the root node
- Parent/child link is done via indexed columns

Popular Alternative

```
select *
from (select *
      from TreeNode
      where and Tree_oid = :1)
connect by TreeNode_rfk = prior TreeNode_oid
start with TreeNode_rfk is null
```

- Internally rewritten to:

```
select *
from TreeNode
connect by TreeNode_rfk = prior TreeNode_oid
           and Tree_oid = :1
start with TreeNode_rfk is null
           and Tree_oid = :1
```

Joins (1)

- Seemingly direct approach:

```
Select nvl(r.fullName_tx,o.UnitName_tx) childName_tx,
       d.*
from TreeNode d,
     Person r,
     OrgUnit o
where d.Person_oid = r.Person_oid (+)
and    d.OrgUnit_oid = o.OrgUnit_oid (+)
and    (d.Person_oid is null or d.role_cd = 'Primry')
connect by d.TreeNode_rfk = prior d.TreeNode_oid
start with d.TreeNode_rfk is null and d.Tree_oid = :1
```

Problems:

- Joins are applied BEFORE hierarchical walk-down
- Other parts of WHERE-clause – afterwards → unnecessary calls!

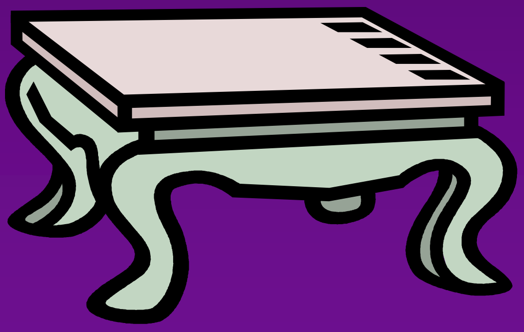
Joins (2)

```
Select nvl(r.fullName_tx,o.UnitName_tx) childName_tx,
       d.*
from
   (select *
    from TreeNode
    connect by TreeNode_rfk = prior TreeNode_oid
    start with TreeNode_rfk is null and Tree_oid = :1) d,
    Person r,
    OrgUnit o
where d.Person_oid = r.Person_oid (+)
and    d.OrgUnit_oid = o.OrgUnit_oid (+)
and    (d.Person_oid is null or d.role_cd = 'Primary')
```

Advantage:

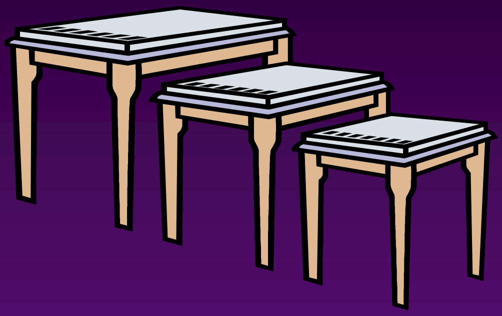
- Clear separation of recursive and non-recursive structure

Common Table Expressions (CTE)



Common Table Expressions

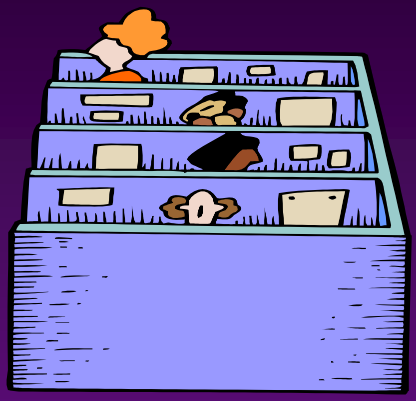
- Oracle’s way of working with recursions
  - CONNECT BY
  - Not part of standard SQL
  - Supported only by small number of other vendors



- Everybody else
  - “Common Table Expressions” (CTE)
  - Part of standard SQL
  - Supported by a number of vendors (SQL Server, MySQL, PostgreSQL)
- Surprise
  - Oracle version 11g R2 - same mechanism introduced as “Recursive Sub-query Factoring”

CTE Example

```
With employees (empno, name, mgr) as
 (
 -- anchor
 select empno, ename, mgr
  from   emp
  where  mgr is null
  union all
 -- recursive block
  select e.empno, e.ename, e.mgr
  from   emp e,
         employees m
  where m.empno = e.mgr
 ) search depth first by name set seq
select empno, name, mgr, seq
from   employees
```



How does it work?

- Concept:
  - Run the anchor part of the UNION ALL query to get root elements.
  - Pass a set of root elements to the second part of the query and get the next set (second level) of records
  - Repeat step 2 until no rows are accessed.
- Advantage:
  - CTE works by SETs of rows, while CONNECT-BY works row-by-row
- Practical aspect (in theory):
  - Should significantly improve performance
  - Higher level of flexibility in your SQL statements.

CTE vs. CONNECT BY

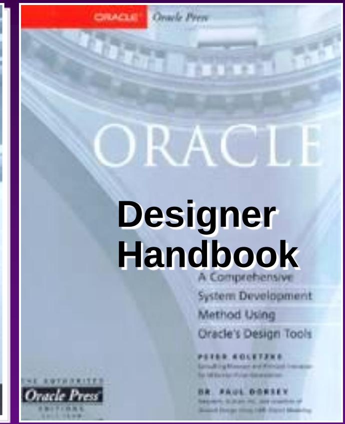
- CTE
  - Can do everything that can be done by CONNECT BY, while the reverse statement is not true.
  - Provides “on the fly” calculation of results, while CONNECT-BY needs everything to be pre-calculated.
- CONNECT-BY
  - Very well optimized. CBO can build much more efficient plans for it.
  - CONNECT-BY has built-in functions
- Keep in mind:
  - In SQL Server, the engine is doing row-level processing “under the hood”! ➔ the real behavior may still be different.
- Conclusion:
  - There is no good reason to use CTE at the current level of implementation.

Conclusions

- Developers should not be afraid of hierarchical data and coding structures!
- It is possible to effectively use them to solve real-life problems.
- The Oracle RDBMS environment is sometimes too rich to blindly make architectural decisions.
- Understanding all of the existing built-ins and ways of internal query optimization can save you from reinventing the wheel.

Contact Information

- Dr. Paul Dorsey – paul_dorsey@dulcian.com
- Michael Rosenblum – mrosenblum@dulcian.com
- Dulcian website - www.dulcian.com



Latest book:

Oracle PL/SQL for Dummies