Creating and Managing Recursive Structures

Andrew Morgan

Analytical Logic cc

Johannesburg, South Africa

October 2005

Andrew.Morgan@telkomsa.net

Abstract

This paper describes how to create recursive structures (self-referencing tables) and how to manipulate these structures using their natural counterpart: recursive stored procedures.

Recursive structures are extremely powerful when used to describe systems of arbitrary complexity. Like a directory tree, they can be made as simple or complex as the user wishes without any impact on the database metadata. Even more useful is the fact that nodes of a given tree may be reclassified within the tree without affecting the referential integrity of the associated data.

Two types of recursions are considered: master and slave. Master recursions have no master relationship and normally have a single root node, although multiple roots are possible. Slave recursions have a root node per master relationship.

Table of Contents

1 Preliminaries....3
1.1 Terminology....3
1.2 Relationships....3
1.3 Root Node....4
1.4 Sample Data....4
2 Slave Recursions....6
3 Basic Stored Procedures....7
3.1 Find a Root....7
3.2 Add a Child....7
3.3 Paste a Child....7
3.4 Rename a Child....8
4 Recursive Stored Procedures....9
4.1 Drill Up....9
4.2 Drill Down....9
4.2.1 Basic Drill Down....9
4.2.2 Enhanced Drill Down....11
4.3 Duplicate Structure....12
4.4 Delete Structure....13
5 Using the Recursions....14
5.1 Design of SpaceMan....14
5.1.1 TMachine....15
5.1.2 TDrive....15
5.1.3 TScan....15
5.1.4 TFolder....16
5.1.5 TFile....16
5.2 Size Rollup....17
5.3 Directory Comparison....18
5.4 Performance Data....21

Table of Illustrations

Illustration 1: High level schematic of several one-to-many relationships....3
Illustration 2: High level schematic of a master recursion....3
Illustration 3: High level schematic of a slave recursion....6
Illustration 4: High level design of SpaceMan....14

1 Preliminaries

1.1 Terminology

Many terms are used interchangeably in database literature, so the following clarification is necessary. When referring to regular tables, individual rows are referred to as such, but rows of a recursion table are often referred to as nodes. Similarly, regular tables are referred to as such, but recursive tables are often referred to as trees.

Parent: the master table/row of a master/detail relationship

Child: the detail table/row a master/detail relationship

Field: equivalent to a column

Attribute: a non-key field

Alternate Key: a candidate key of the table implemented as a unique index

Leaf: a node with no children

Branch: a node with at least one child node (Note that when a branch has had all its leaves deleted, it automatically becomes a leaf)

Root: a special case of a branch

Node: a row in a recursive table; either a branch or a leaf

Tree: all rows belonging ultimately to a root node (for a single-root table, this is all the rows of the table)

1.2 Relationships

When one considers typical database relationships, they are of the form one-to-one or one-to-many. Of these, the one-to-many is the most useful and is depicted below:

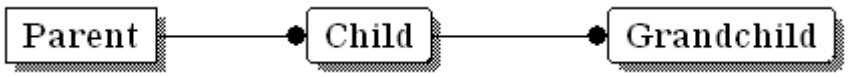

Illustration 1: High level schematic of several one-to-many relationships

Each link points to the parent row in the master table. The difficulty with designing systems of arbitrary complexity with this strategy is that the number of levels (linked tables) must be known in advance which by definition is not. Changing the number of levels is no trivial matter.

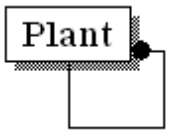

Illustration 2: High level schematic of a master recursion

Using a recursion solves this problem. The child nodes exist in the same table as the parent. Each link points to the parent node. The child nodes may themselves have

children. The number of levels that can be implemented is virtually limitless: each node must simply identify its parent.

One of the most notable differences between these two schemes is that the regular design may include specific attributes about the Parent, Child and Grandchild tables, whilst the recursion does not. These are normally implemented as sub-types. The minimum requirements necessary for a recursion to operate are:

- Primary Key: the unique key for the node, generically referred to as *ID*
- Foreign Key: the key of the parent node, generically referred to as *PID* (Parent ID)
- Node text: the text to display in a tree control, generically referred to as *Name*

The nature of the PK and FK are important for performance reasons and need to be carefully considered. Coupled with this is the problem of determining the alternate key for the tree. Good choices are usually (*Name* – *Name* is globally unique) or (*PID*,*Name* – *Name* is unique to a particular branch). After a great deal of experimentation, the best choice for the keys are INTEGER.

The design philosophy which I use is formally called "Identifying Surrogate Key Design" [1]. In this philosophy, we proceed as follows:

The table is assigned a three-letter mnemonic (*PLA*), and this prefixes all the column names and index/constraint names. The primary key of the table will be *plaID*, and it will have a rolenamed foreign key (*plaPID*) to itself. The alternate key strategy will be (*plaPID, plaName*), that is, within a particular branch, the node names must be unique:

```
CREATE TABLE Plant (
          plaID     INTEGER NOT NULL,
          plaPID    INTEGER NOT NULL,
          plaName   VARCHAR(64) NOT NULL);
CREATE UNIQUE INDEX PLA_AK1 ON Plant(plaPID,plaName);
ALTER TABLE Plant ADD CONSTRAINT PLA_PK PRIMARY KEY(plaID);
ALTER TABLE Plant ADD CONSTRAINT PLA_PLA FOREIGN KEY(plaPID) REFERENCES Plant;
```

1.3 Root Node

In order to start a recursion, a root node is required and a consistent strategy needs to be used to identify them, especially for multi-root tables. Using a NULL for *PID* seemed a candidate, but the flaw with this is that the referential integrity becomes dangerously relaxed. Using a constant is also a possibility (eg -1), but a node with key -1 must exist, and so becomes a super-root. The best solution is to use the same value for the *PID* and *ID*; that is, the root is the self-owned node (much like a self-signed certificate). Conveniently, many roots are thus possible.

In earlier versions of Firebird, it was not possible to execute:

```
INSERT INTO Plant(plaID,plaPID,plaName) VALUES(0,0,'Eskom');
```

since the foreign key entry did not previously exist, however, this is no longer a problem.

1.4 Sample Data

plaID	plaPID	plaName	Comment
0	0	Eskom	Root – the self-owned node
1	0	Transmission	Owned by Eskom
2	1	Koeberg	Owned by Transmission
3	1	Tutuka	Owned by Transmission
4	0	Generation	Owned by Eskom
5	4	Koeberg	Owned by Generation
6	4	Matla	Owned by Generation

Primary Key
Alternate Key

Arranged hierarchically, this looks like:

Eskom

- Transmission
  - Koeberg
  - Tutuka
- Generation
  - Koeberg
  - Matla

Notice that Koeberg appears twice which is allowable with the chosen alternate key strategy.

A new root could now be added as follows:

```
plaID     = GEN_ID(Plant,1);
INSERT    INTO Plant(plaID,plaPID,plaName)
          VALUES(:plaID,:plaID,'Organization');
```

2 Slave Recursions

It is possible to implement slave recursions which are defined as single root node per relationship. The master table of the relationship could be a regular table or a recursion as depicted below:

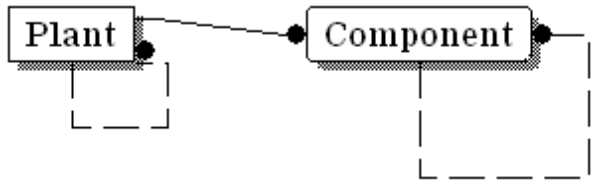

Illustration 3: High level schematic of a slave recursion

This design shows a plant recursion which is parent to a component recursion. The reasons for splitting a recursion like this vary, but in this case, additional data will be stored about components, and they will also have sampled values, which a plant would not have. The alternate key strategy for the *Component* table again requires consideration, and here a good choice is (*plaID,comName*). This is interpreted as a unique component per plant, but multiple instances of that component across different plants.

```
CREATE TABLE Component (
          plaID     INTEGER NOT NULL,
          comID     INTEGER NOT NULL,
          comPID    INTEGER NOT NULL,
          comName   VARCHAR(64) NOT NULL);
CREATE UNIQUE INDEX COM_SK ON Component(comID);
CREATE UNIQUE INDEX COM_AK1 ON Component(plaID,comName);
ALTER TABLE Component ADD PRIMARY KEY (plaID,comID);
ALTER TABLE Component ADD FOREIGN KEY (plaID) REFERENCES Plant;
ALTER TABLE Component ADD FOREIGN KEY (plaID,comPID) REFERENCES Component;
```

There are several important things to notice with the above definition:

- As a result of the identifying nature of the design, the primary key for the *Component* table is (*plaID,comID*). It is necessary to distinguish another important key which is *COM_SK* (Surrogate Key), because *comID* is unique in its own right.
- The *Component* foreign key back to itself only provides a rolename for the lowest level of the composite key (ie *comPID*).
- Plant does not have a *PLA_SK* index because it would have the same definition as *PLA_PK*. This is true of top level tables. Slave recursions must implement *XXX_SK*.

3 Basic Stored Procedures

In order to efficiently navigate around the recursive tables, a number of stored procedures are required to fulfill the following tasks. These are split into two categories:

Basic stored procedures:

- Find a root
- Add a child
- Paste a child (re-classification)
- Rename a child

and recursive stored procedures:

- Drill up through the structure
- Drill down through the structure
- Duplicate a portion of the structure
- Delete a portion of the structure

The basic stored procedures will be covered in this section, and recursive in the next.

3.1 Find a Root

Typically, find a root would be specific to a table. In the example above, the Plant table has a single root which could be found by:

```
SELECT    plaID
FROM      Plant
WHERE     plaID     = plaPID;
```

The Component table is more interesting:

```
SELECT    comID
FROM      Component
WHERE     comID     = comPID
AND       plaID     = :plaID;
```

and requires that a specific plant (the master relationship) be supplied as a parameter.

3.2 Add a Child

Add a leaf to a specified parent node.

```
comID     = GEN_ID(Component,1);
INSERT    INTO Component(plaID,comID,comPID,comName)
          VALUES(:plaID,:comID,:comPID,:comName);
```

The plant (*plaID*), parent node (*comPID*) and text (*comName*) would be supplied as parameters. It is useful to return *comID*.

3.3 Paste a Child

This moves a node to a different parent.

```
UPDATE    Component SET
          comPID    = :comPID
WHERE     comID     = :comID;
```

Note that because *comPID* is not part of the primary key, this update has no effect on the referential integrity relating to Component. Thus, nodes can be very easily re-classified!

3.4 Rename a Child

```
UPDATE    Component SET
          comName   = :comName
WHERE     comID     = :comID;
```

Note that because of the identifying surrogate key design, *comName* is not part of the primary key, this update has no effect on the referential integrity relating to Component.

4 Recursive Stored Procedures

To recap, the following stored procedures are covered:

- Drill up through the structure
- Drill down through the structure
- Duplicate a portion of the structure
- Delete a portion of the structure

4.1 Drill Up

The first of the challenging procedures. This is not strictly a recursive procedure, because it does not call itself, but rather iterates through an internal loop. Given a particular *Node*, drill up the given number of levels (*Height*), or until exhaustion.

```
ALTER     PROCEDURE PlantUp(Node INTEGER,Height INTEGER)
RETURNS   (plaID    INTEGER,
           plaPID   INTEGER,
           plaName  VARCHAR(64),
           plaLevel INTEGER) AS
BEGIN
WHILE     (Height != 0) DO
          BEGIN
          SELECT   plaID,plaPID,plaName,:Height
          FROM     Plant
          WHERE    plaID = :Node
          INTO     :plaID,:plaPID,:plaName,:plaLevel;
          SUSPEND;
          Node     = plaPID;
          Height   = Height - 1;
          IF       (plaID = plaPID)
          THEN     EXIT;
          END
END#
```

This procedure locates the required node and returns it. It then alters the *Node* parameter to the current node's parent and decrements *Height* for the next iteration. The procedure terminates after returning the requested levels, or *plaID* = *plaPID* (the root has been found and every subsequent iteration would return the root). Note the *plaLevel* parameter which is automatically supplied. Note also that supplying a negative *Height* forces the procedure to drill up exhaustively.

4.2 Drill Down

The requirement of this procedure is to drill out a section of the tree from a particular node, for a number of levels. There are many ways to go about the recursion, but it is best to develop the ideas and then enhance them.

4.2.1 Basic Drill Down

```
ALTER     PROCEDURE PlantDown(Node INTEGER,Depth INTEGER)
RETURNS   (plaID    INTEGER,
           plaPID   INTEGER,
           plaNAME  VARCHAR (64),
           plaLevel INTEGER) AS
DECLARE   VARIABLE Child INTEGER;
BEGIN
--        Stage 1 (Return the current Node)
--        PLAN (PLANT INDEX (PLA_PK))
SELECT    plaID,plaPID,plaName,-:Depth
FROM      Plant
WHERE     plaID = :Node
INTO      :plaID,:plaPID,:plaName,:plaLevel;
SUSPEND;
IF        (Depth = 0)
THEN      EXIT;
Depth     = Depth - 1;
--        Stage 2 (Create a list of the Node's children)
--        PLAN SORT ((PLANT INDEX (PLA_AK1,PLA_PLA)))
FOR       SELECT    plaID
          FROM      Plant
          WHERE     plaPID = :Node
          AND       plaPID != plaID
          ORDER     BY plaName
          INTO      :Child DO
          BEGIN
          --        Stage 3 (Drill out the Node's children recursively)
          FOR       SELECT    plaID,plaPID,plaName,plaLevel
                    FROM      PlantDown(:Child,:Depth)
                    INTO      :plaID,:plaPID,:plaName,:plaLevel DO
                    BEGIN
                    SUSPEND;
                    END
          END
END#
```

This procedure consists of three main stages.

- Stage 1 returns the current node. If depth is zero, the drill down has proceeded as far as needed (no children of the current node are required).
- Stage 2 creates a list of the current node's children. Note the condition *plaPID != plaID* which is necessary to prevent the root node from endlessly returning itself as a child! This condition is a recurring theme for most of the recursive procedures.
- Stage 3 drills out each of the children, by recursively calling *PlantDown*.

Although functional, this procedure is rather slow because stage 1 returns a single node on each execution. It would be preferable to return all the children of a node in a single select. However, if the procedure returns only children, the problem is to return the initial node.

4.2.2 Enhanced Drill Down

In order to return the initial node, and get all children in a single select, the procedure needs to be split into two. Stage 1 goes into one procedure, and stages 2 and 3 go into a second:

```
ALTER     PROCEDURE PlantDown(Node INTEGER,Depth INTEGER)
RETURNS   (plaID    INTEGER,
           plaPID   INTEGER,
           plaName  VARCHAR (64),
           plaLevel INTEGER) AS
BEGIN
--         Return the current Node
--         PLAN (PLANT INDEX (PLA_PK))
SELECT     plaID,plaPID,plaName,-:Depth
FROM       Plant
WHERE      plaID = :Node
INTO       :plaID,:plaPID,:plaName,:plaLevel;
SUSPEND;
IF         (Depth = 0)
THEN       EXIT;
Depth      = Depth - 1;
FOR        SELECT   plaID,plaPID,plaName,plaLevel
           FROM     PlantDrill(:Node,:Depth)
           INTO     :plaID,:plaPID,:plaName,:plaLevel DO
           BEGIN
           SUSPEND;
           END
END#
```

This procedure returns the initial node, and sets up the recursion. The test for zero depth is necessary to prevent an exhaustive drill down.

```
ALTER     PROCEDURE PlantDrill(Node INTEGER,Depth INTEGER)
RETURNS   (plaID    INTEGER,
           plaPID   INTEGER,
           plaName  VARCHAR(64),
           plaLevel INTEGER) AS
DECLARE   VARIABLE Remain INTEGER;
DECLARE   VARIABLE Child INTEGER;
BEGIN
Remain     = Depth - 1;
--         Stage 1 (Create a list of the Node's children)
--         PLAN SORT ((PLANT INDEX (PLA_AK1,PLA_PLA)))
FOR        SELECT   plaID,plaPID,plaName,-:Depth
           FROM     Plant
           WHERE    plaPID    = :Node
           AND      plaPID    != plaID
           ORDER    BY plaName
           INTO     :plaID,:plaPID,:plaName,:plaLevel DO
           BEGIN
           SUSPEND;
           --       Stage 2 (Drill out the Node's children recursively)
```

```
        IF      (Depth != 0) THEN
                BEGIN
                Child    = :plaID;
                FOR      SELECT   plaID,plaPID,plaName,plaLevel
                         FROM     PlantDrill(:Child,:Remain)
                         INTO     :plaID,:plaPID,:plaName,:plaLevel DO
                         BEGIN
                         SUSPEND;
                         END
                END
        END
END#
```

This procedure consists of two main stages:

- Stage 1 returns ***all*** the children of the current Node.
- Stage 2 executes if the depth is non-zero (more levels are required). It recursively calls *PlantDrill* for the current child node.

Note the *plaLevel* parameter which is automatically supplied – this will be very useful later on. Note also that supplying a negative *Depth* forces the procedure to drill down exhaustively.

In order by build a fast visual component, additional enhancements are required. Specifically, a tree component would always drill down 2 levels (in response to clicking the +). The first level supplies all the children, and the second supplies grandchildren, so that the button state would be correct (ie whether or not a + appears next to each child). Since only one grandchild is necessary to establish a + the procedure should be able to return a reduced dataset. This is left as an exercise for the reader.

4.3 Duplicate Structure

```
ALTER    PROCEDURE PlantCopy(Source INTEGER,Target INTEGER) AS
DECLARE  VARIABLE SID INTEGER;
DECLARE  VARIABLE TID INTEGER;
DECLARE  VARIABLE Name VARCHAR(64);
BEGIN
FOR      SELECT   GEN_ID(Plant,1),plaID,plaName
         FROM     Plant
         WHERE    plaPID = :Source
         AND      plaPID != plaID
         INTO     :TID,:SID,:Name DO
         BEGIN
         INSERT   INTO TPlant(plaID,plaPID,plaName)
         VALUES   (:TID,:Target,:Name);
         EXECUTE  PROCEDURE PlantCopy(SID,TID);
         END
END#
```

This turns out to be a remarkably simply procedure. The parameters indicate the *Source* and *Target* nodes for the copy. The select simultaneously:

- creates a list of the *Source* node's children
- generates a key for the new node

- sets up the recursion by identifying the next level *Source* (*SID*) and *Target* (*TID*).

Each child node is inserted using *Target* as the *PID*, and the procedure is recursively called.

4.4 Delete Structure

```
ALTER     PROCEDURE PlantDelete(ID INTEGER) AS
BEGIN
FOR       SELECT    plaID
          FROM      PlantDown(:ID,-1)
          ORDER     BY plaLevel DESC
          INTO      :plaID DO
          BEGIN
          DELETE    FROM      TPlant
          WHERE     plaID     = :plaID;
          END
END#
```

This also turns out to be a remarkably simply procedure. It is strictly not a recursion, but sources its data from a recursion. The difficulty with deleting nodes is that the top node cannot be deleted since the presence of child nodes will violate the referential integrity. The solution is to drill out the tree to exhaustion with *PlantDown* and invert it! This way the tree is deleted from the bottom up, leaf-by-leaf.

5 Using the Recursions

Having covered the primitives for manipulating the recursion, an examination of a working application would be instructive.

Directory structures are familiar to everyone using computers, and most would readily identify with them.

The objective of this application is to compute the total storage of a particular folder by summing the size its files and recursively, the size of its sub-folders. The usage is then ranked by the portion that the file or folders uses.

I originally wrote an application to do this using live scans and it worked fine, so converting it to a database application was purely for this demonstration. However, the gains from doing this have been great as we shall see.

5.1 Design of SpaceMan

Instead of live scans, the application now performs a drive snapshot and stores it in the database. Already there is a performance gain because subsequent live scans are no longer required. As time goes by, new snapshots need to be performed, and so the system accumulates a series. The huge gain now is that scans may be compared, and not just with themselves. A scan of one drive may be compared to a scan of a drive on another machine (whose snapshot has been stored). This may be useful for comparing a network of machines to a master structure.

Since the comparisons will be performed recursively, it is possible to compare just portions of the structure. For example, one may compare C:\WINDOWS to D:\WIN on a different machine to detect changes to OS files only.

The application is designed to process a standard redirected directory listing, so the snapshots may be performed locally, and sent to a central machine for processing. Consequently, it is possible to perform scans of removable media and mapped drives, however these may be neither useful nor efficient.

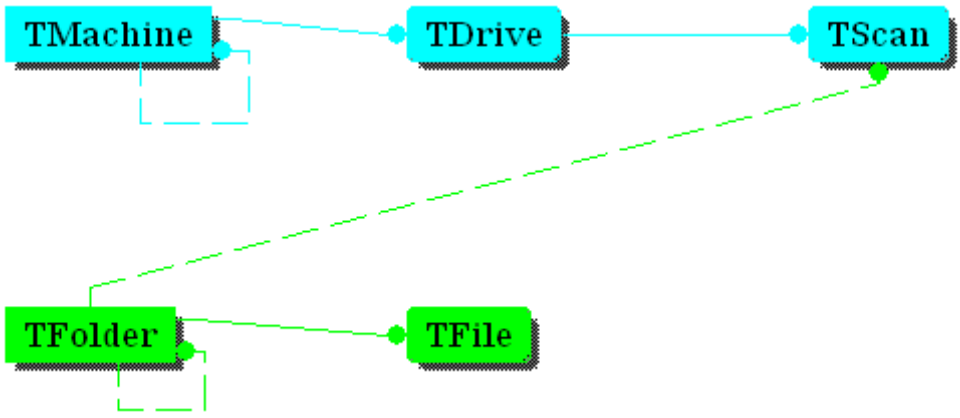

Illustration 4: High level design of SpaceMan

The design in a nutshell has a machine single-root master recursion. Each machine has many drives. Each drive has many scans (snapshots).

Folder is a muli-root master recursion. Each root represents a single scan and is stored in the scan table. Each folder has many files.

5.1.1 TMachine

This table is a single-root master recursion and stores all the machines in a tree. The intention of the tree is to be able to arbitrarily classify the machines. For example, the root could be the domain, the next level could be routers and the final level the actual computers. Alternatively, the tree could be arranged by department. The point is that the classification is arbitrary, and may be changed without affecting the referential integrity.

```
CREATE    TABLE     TMachine(
          macID     DSURROGATE NOT NULL,
          macPID    DSURROGATE NOT NULL,
          macName   DSTRING32 NOT NULL);
ALTER     TABLE     TMachine
          ADD       CONSTRAINT MAC_PK PRIMARY KEY(MACID);
ALTER     TABLE     TMachine
          ADD       CONSTRAINT MAC_MAC FOREIGN KEY(MACPID)
                    REFERENCES TMachine(MACID);
CREATE    UNIQUE    INDEX MAC_AK ON TMachine(MACNAME);
```

The chosen alternate key strategy is globally unique names.

5.1.2 TDrive

This is a regular table.

```
CREATE    TABLE     TDrive(
          macID     DSURROGATE NOT NULL,
          driID     DSURROGATE NOT NULL,
          driName   DSTRING32 NOT NULL);
ALTER     TABLE     TDrive
          ADD       CONSTRAINT DRI_PK PRIMARY KEY(MACID,DRIID);
ALTER     TABLE     TDrive
          ADD       CONSTRAINT DRI_MAC FOREIGN KEY (MACID) REFERENCES
                    TMachine(MACID);
CREATE    UNIQUE    INDEX DRI_SK ON TDrive(DRIID);
CREATE    UNIQUE    INDEX DRI_AK ON TDrive(MACID,DRINAME);
```

The chosen alternate key strategy for this table is (*macID,driName*), ie a particular drive letter may only appear once per machine. Notice the surrogate key index *DRI_SK*.

5.1.3 TScan

This is a regular table.

```
CREATE    TABLE     TScan(
          macID     DSURROGATE NOT NULL,
          driID     DSURROGATE NOT NULL,
          scaID     DSURROGATE NOT NULL,
          folID     DSURROGATE NOT NULL,
          scaName   DDATETIME NOT NULL);
ALTER     TABLE     TScan
```

```
          ADD       CONSTRAINT SCA_PK PRIMARY KEY(MACID,DRIID,SCAID);
ALTER     TABLE     TScan
          ADD       CONSTRAINT SCA_DRI FOREIGN KEY(MACID,DRIID)
                    REFERENCES TDrive(MACID,DRIID);
ALTER     TABLE     TScan
          ADD       CONSTRAINT SCA_FOL FOREIGN KEY(FOLID)
                    REFERENCES TFolder(FOLID);
CREATE    UNIQUE    INDEX SCA_SK ON TScan(SCAID);
CREATE    UNIQUE    INDEX SCA_AK ON TScan(FOLID);
```

The chosen alternate key strategy for this table is (*folID*). The foreign key points to the root node of a scan which may only be referenced once. Notice the surrogate key index *SCA_SK*.

5.1.4 TFolder

This table is a multi-root master recursion and exhaustively stores all the folders of the drive being scanned. Although it has the same capabilities as the machine table, this is essentially a read-only table.

```
CREATE    TABLE     TFolder(
          folID     DSURROGATE NOT NULL,
          folPID    DSURROGATE NOT NULL,
          folSize   DDOUBLE NOT NULL,
          folName   DSTRING128 NOT NULL);
ALTER     TABLE     TFolder
          ADD       CONSTRAINT FOL_PK PRIMARY KEY(FOLID);
ALTER     TABLE     TFolder
          ADD       CONSTRAINT FOL_FOL FOREIGN KEY(FOLPID)
                    REFERENCES TFolder(FOLID);
CREATE    UNIQUE    INDEX FOL_AK ON TFolder(FOLPID,FOLNAME);
```

The chosen alternate key strategy for this table is (*folPID,folName*). This is consistent with the real world problem being modelled: a folder name cannot be repeated within a particular folder, but may otherwise appear any number of times on the drive. This is a top level table, and so the surrogate key is omitted, being identical to the primary key.

Notice *folSize* implemented as a double because of file sizes in excess of 32-bit[1]. This attribute is a placeholder for subsequent roll-ups.

5.1.5 TFile

```
CREATE    TABLE     TFile(
          folID     DSURROGATE NOT NULL,
          filID     DSURROGATE NOT NULL,
          filSize   DDOUBLE NOT NULL,
          filName   DSTRING128 NOT NULL);
ALTER     TABLE     TFile
          ADD       CONSTRAINT FIL_PK PRIMARY KEY(FOLID,FILID);
ALTER     TABLE     TFile
          ADD       CONSTRAINT FIL_FOL FOREIGN KEY(FOLID)
                    REFERENCES TFolder(FOLID);
```

1 A future upgrade to BIGINT may be preferable

```
CREATE    UNIQUE    INDEX FIL_SK ON TFile(FILID);
CREATE    UNIQUE    INDEX FIL_AK ON TFile(FOLID,FILNAME);
```

The chosen alternate key strategy for this table is (*folID*,*filName*). This is consistent with the real world problem being modelled: a file name cannot be repeated within a particular folder, but may otherwise appear any number of times on the drive. Notice the surrogate key index *FIL_SK*.

Notice *filSize* implemented as a double because of file sizes in excess of 32-bit[2]. This attribute is set while inserting from the directory listing.

Those who a sharp will notice one tiny flaw in the Folder/File part of the design: a folder named X and a file named X may appear within a particular folder. However, since the file table is also essentially read-only and is populated from a directory listing this problem may be ignored.

5.2 Size Rollup

Now the power of a database and recursions really come to bear. The task remains to sum the size of all the files and folders and update the size of the corresponding folder. Note that slack space is currently ignored[3].

```
ALTER     PROCEDURE PRollSize(folPID INTEGER) AS
DECLARE   folID     INTEGER;
DECLARE   folSize   DOUBLE PRECISION;
DECLARE   filSize   DOUBLE PRECISION;
BEGIN
--        Invert the tree and go!
FOR       SELECT    folID
          FROM      PFolderDown(:folPID,-1,-1)
          ORDER     BY folLevel DESC
          INTO      :folID DO
          BEGIN
          --        Sum the size of files belonging to this folder.
          SELECT    SUM(filSize)
          FROM      TFile
          WHERE     folID     = :folID
          INTO      :filSize;

          --        Sum the size of sub-folders belonging to this folder
          SELECT    SUM(folSize)
          FROM      TFolder
          WHERE     folPID    = :folID
          AND       folID     != folPID
          INTO      :folSize;

          --        Update!
          UPDATE    TFolder   SET
                    folSize   = COALESCE(:folSize,0) + COALESCE(:filSize,0)
          WHERE     folID     = :folID;
```

2 A future upgrade to BIGINT may be preferable.
3 A future upgrade to compute slack space at file level may be advantageous.

```
        END
END
```

The trick to the rollup is to invert the folder listing. This forces the summation to begin at the folder leaves (folders without sub-folders) and proceed up to the root. Notice the *folID != folPID* condition which prevents the size of the root being added to the size of it children (this is not a problem on the first execution, but each time the rollup is executed, the root would grow by a multiple of its original size).

5.3 Directory Comparison

One of the great benefits of having stored the snapshots is to compare them. The output of this procedure is a list a files with relative path and file size information. At first sight this is a very complex procedure but in reality the original block has been split into three optimized blocks which are easy to grasp.

```
CREATE    PROCEDURE PFolderCompare (
          Parent    VARCHAR (255),
          oldID     INTEGER,
          newID     INTEGER)
RETURNS   (Directory         VARCHAR (255),
          Filename  VARCHAR (255),
          Status    VARCHAR (16),
          OldSize   DOUBLE PRECISION,
          NewSize   DOUBLE PRECISION) AS
DECLARE   VARIABLE folID1 INTEGER;
DECLARE   VARIABLE folID2 INTEGER;
BEGIN
--        Deleted (Folders in old missing in new)
FOR       SELECT    :Parent || '\' || F1.folName, F1.folID, -1
          FROM      TFolder F1
          LEFT      JOIN TFolder F2
                    ON        F2.folPID = :newID
                    AND       F2.folPID != F2.folID
                    AND       F1.folName = F2.folName
          WHERE     F1.folPID = :oldID
          AND       F1.folPID != F1.folID
          AND       F2.folID IS NULL
          INTO      :Directory, folID1, :folID2 DO
          BEGIN
          FOR       SELECT    filName, 'Deleted', filSize, -1
                    FROM      TFile
                    WHERE     folID = :folID1
                    INTO      :Filename, :Status, :OldSize, :NewSize DO
                    BEGIN
                    SUSPEND;
                    END
          FOR       SELECT    Directory, Filename, Status, OldSize, NewSize
                    FROM      PFolderCompare(:Directory,:folID1,:folID2)
                    INTO      :Directory, :Filename, :Status, :OldSize, :NewSize
                              DO
                    BEGIN
```

```
                SUSPEND;
                END
        END

--      Changed (Folders present in old and new)
FOR     SELECT  :Parent || '\' || F1.folName, F1.folID, F2.folID
        FROM    TFolder F1
        JOIN    TFolder F2
                ON      F1.folName = F2.folName
        WHERE   F1.folPID = :oldID
        AND     F1.folPID != F1.folID
        AND     F2.folPID = :newID
        AND     F2.folPID != F2.folID
        INTO    :Directory, folID1, :folID2 DO
        BEGIN
        FOR     SELECT  Filename, Status, OldSize, NewSize
                FROM    PFileCompare(:folID1,:folID2)
                INTO    :Filename, :Status, :OldSize, :NewSize DO
                BEGIN
                SUSPEND;
                END
        FOR     SELECT  Directory, Filename, Status, OldSize, NewSize
                FROM    PFolderCompare(:Directory,:folID1,:folID2)
                INTO    :Directory, :Filename, :Status, :OldSize, :NewSize
                        DO
                BEGIN
                SUSPEND;
                END
        END

--      Added (Folders present in new missing in old)
FOR     SELECT  :Parent || '\' || F2.folName, -1, F2.folID
        FROM    TFolder F1
        RIGHT   JOIN TFolder F2
                ON      F1.folPID = :oldID
                AND     F1.folPID != F1.folID
                AND     F1.folName = F2.folName
        WHERE   F2.folPID = :newID
        AND     F2.folPID != F2.folID
        AND     F1.folID IS NULL
        INTO    :Directory, folID1, :folID2 DO
        BEGIN
        FOR     SELECT  filName, 'Added', -1, filSize
                FROM    TFile
                WHERE   folID = :folID2
                INTO    :Filename, :Status, :OldSize, :NewSize DO
                BEGIN
                SUSPEND;
                END
        FOR     SELECT  Directory, Filename, Status, OldSize, NewSize
```

```
                    FROM      PFolderCompare(:Directory,:folID1,:folID2)
                    INTO      :Directory, :Filename, :Status, :OldSize, :NewSize
                              DO
                    BEGIN
                    SUSPEND;
                    END
          END
END
```

This procedure uses the convention of left for old and right for new. Notice that a file size of -1 is used to indicate a missing file. The code consists of three optimized blocks which are covered out of sequence:

- Deleted. These are folders existing in the old scan, but missing in the new. They are located by an ***non-join*** (a left outer join with null entries on the right). Since the files belonging to new are known to be missing, the file list is simply a select of files for the old folder.
- Added. These are folders existing in the new scan, but missing in the old. They are located by an ***non-join*** (a right outer join with null entries on the left). Since the files belonging to old are known to be missing, the file list is simply a select of files for the new folder.
- Changed. These are folders that exist in both scans and are located by an inner join. As to differences in files, the situation is not as simple as for Deleted or Added. A helper procedure is used to perform this comparison (*PFileCompare*).

Each block recursively calls *PFolderCompare* to drill through the directory structure to exhaustion.

Like *PFolderCompare*, *PFileCompare* at first sight this is a very complex procedure but in reality the original block has been split into three optimized blocks which are easy to grasp.

```
ALTER     PROCEDURE PFileCompare(
          oldID INTEGER,
          newID INTEGER)
RETURNS   (Filename VARCHAR (255),
          Status    VARCHAR (16),
          OldSize   DOUBLE PRECISION,
          NewSize   DOUBLE PRECISION) AS
BEGIN
--        Deleted.
FOR       SELECT    F1.filName, 'Deleted', F1.filSize, -1
          FROM      TFile F1
          LEFT      JOIN TFile F2
                    ON F2.folID = :newID
                    AND F1.filName = F2.filName
          WHERE     F1.folID = :oldID
          AND       F2.filID IS NULL
          INTO      :FileName, :Status, :OldSize, :NewSize DO
          BEGIN
          SUSPEND;
          END
```

```
--        Changed.
FOR       SELECT    F1.filName, F1.filSize, F2.filSize
          FROM      TFile F1
          JOIN      TFile F2
                    ON F1.filName = F2.filName
          WHERE     F1.folID = :oldID
          AND       F2.folID = :newID
          AND       F1.filSize != F2.filSize
          INTO      :FileName, :OldSize, :NewSize DO
          BEGIN
          IF        (NewSize > OldSize)
                    THEN      Status = 'Bigger';
                    ELSE      Status = 'Smaller';
          SUSPEND;
          END

--        Added.
FOR       SELECT    F2.filName, 'Added', -1, F2.filSize
          FROM      TFile F1
          RIGHT     JOIN TFile F2
                    ON F1.folID = :oldID
                    AND F1.filName = F2.filName
          WHERE     F2.folID = :newID
          AND       F1.filID IS NULL
          INTO      :FileName, :Status, :OldSize, :NewSize DO
          BEGIN
          SUSPEND;
          END
END
```

This procedure uses the convention of left for old and right for new. Notice that a file size of -1 is used to indicate a missing file. The code consists of three optimized blocks which are covered out of sequence:

- Deleted. These are files existing in the old scan, but missing in the new. They are located by an ***non-join*** (a left outer join with null entries on the right).
- Added. These are files existing in the new scan, but missing in the old. They are located by an ***non-join*** (a right outer join with null entries on the left).
- Changed. These are files that exist in both scans and are located by an inner join. Note that only files with a size difference are selected; other files are considered to be the same[4].

5.4 Performance Data

Directory scan: 180s.

Snapshot creation: 150s.

Size rollup: 4s

Entire drive compare: 8.5s

4 A better strategy may be to compute a CRC (cheap) or Hash (expensive) and compare those instead. These would have to be computed by the client machines.

Bibliography

1 Andrew Morgan Database Design with Identifying Surrogate Key (2000)