Paper 1865-2018

Who's Your Daddy: Managing Parent-Child Hierarchies in SAS®

Richard Collins, RWC Technical Solutions LTD

ABSTRACT

Reporting hierarchies underpin most, if not all, business reporting solutions. Accurate hierarchies enable reporting lines to quickly drill into and deal with issues, address key performance indicator (KPI) fluctuations, and of course, receive credit for effort expended. Without an accurate hierarchy, KPI reporting in large organizations would quickly become mistrusted and suffer the dreaded loss of adoption and potentially become extinct. This paper shows you how to simply manage large, multi-level hierarchies, complete with change history, through a simple HTML5 interface built on top of the SAS® Stored Process Web Application. Powered by the Boemska HTML5 Data Adapter for SAS® (H54S) and the built-in hierarchical query functions of major database management systems (DBMSs) (Oracle and PostgreSQL), you can manage your business reporting hierarchy with just a few lines of Base SAS® code. This solution has been proven in production for over 5000 leaf members across 10 levels, supporting approximately 500 reporting line changes per month. Maintaining full change history delivers a full audit trail and historical roll-up functionality, this solution also supports future dated changes to enable your line managers to plan departmental changes well in advance.

INTRODUCTION

This paper will demonstrate how to use SAS to manage data hierarchies by leveraging the powerful capabilities of Base SAS, SAS SQL and SAS ACCESS engines in combination with ISO/ANSI standard DBMS functionality.

The techniques and ideas presented here will provide you with the foundations for building an enterprise ready HTML5 web application powered by the Boemska HTML5 data adapter for SAS, capable of managing large multi-level hierarchical data structures.

All data and code used to demonstrate examples in this paper will be shown in more detail in the accompanying presentation and will be available in the UHM project at the following URL: https://github.com/richardwcollins/uhm

MODELING AN ORGANIZATION HIERARCHY

While many businesses seek a flatter structure, there are still many traditional hierarchical organizations and even for those looking to reduce levels of management; ownership and responsibility are key aspects of business operations which allow processes to be monitored, managed and controlled, especially when implementing enterprise applications.

The adjacency model or parent-child hierarchy is probably the most widely used hierarchy modeling technique. Each node knows its immediate parent, for example an employee's manager.

For purposes of keeping things simple, our sample organization is traditional in structure, with 5 levels of seniority from Managing Director down to Employee.

Figure 1 shows an abridged version of the organizational structure, with the IT department expanded down to employee level:

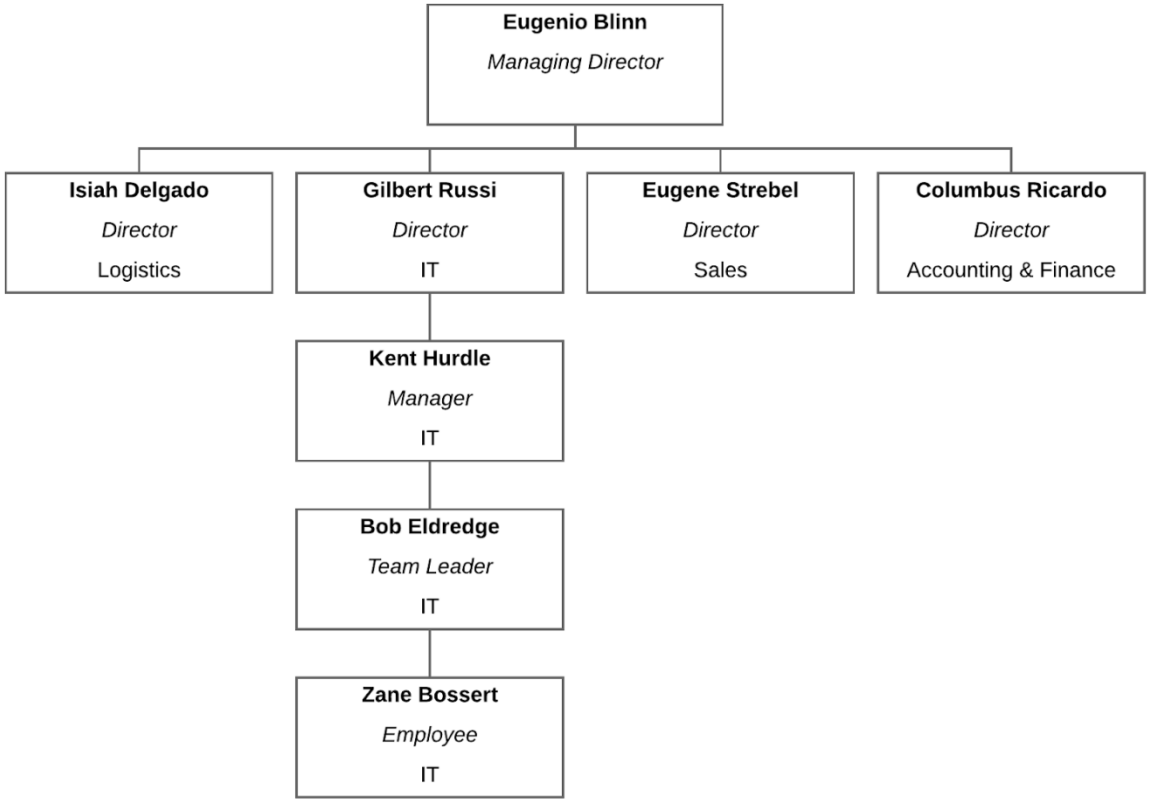


Figure 1 - Abridged organizational hierarchy

THE DATA MODEL

The organizational hierarchy is stored in a user table with each node having its parent ID (or manager/direct report) stored along with it. This is a readily available data source for any business, or can be easily generated based on known information about how a company is structured.

Figure 2 shows the user data model

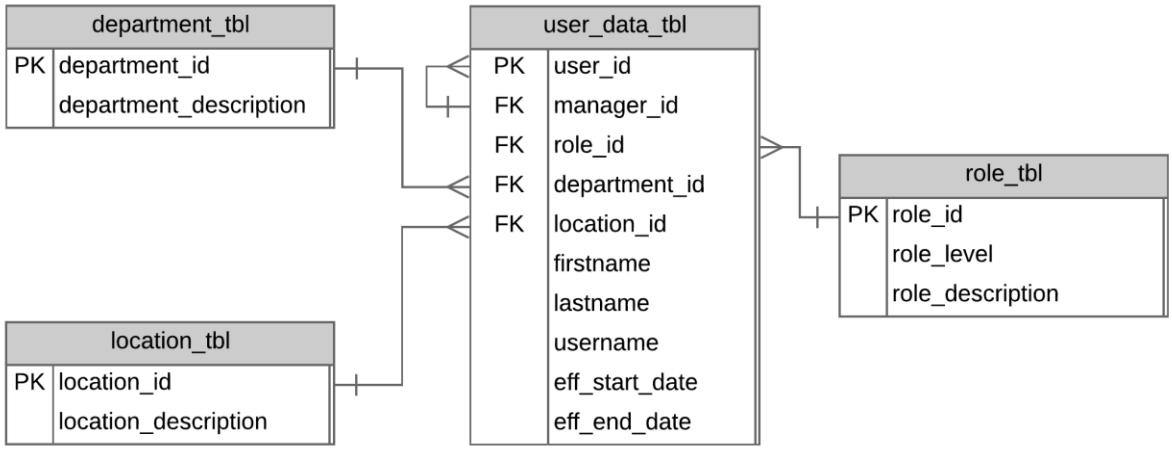


Figure 2 - The user data model

This simple data model can be extended to support additional attributes, either as a column/variable in the user table or by joining additional attribute tables.

A simple methodology for exploring the hierarchy and determining the full tree might be to use a self-join, joining the user table to itself, using aliases as many times as is required to reach the top or bottom of the tree.

For example, to return employees in the IT department and their reporting line up to the managing director as shown in the organization chart, the SQL query would be as follows:

```
SELECT
       t1.firstname||' '|| t1.lastname as lev1,
       t2.firstname||' '|| t2.lastname as lev2,
       t3.firstname||' '|| t3.lastname as lev3,
       t4.firstname||' '|| t4.lastname as lev4,
       t5.firstname||' '|| t5.lastname as lev5
FROM rwcdev.user data tbl AS t1
LEFT JOIN rwcdev.user_data_tbl AS t2 ON t2.manager_id = t1.user_id
LEFT JOIN rwcdev.user_data_tbl AS t3 ON t3.manager_id = t2.user_id
LEFT JOIN rwcdev.user_data_tbl AS t4 ON t4.manager_id = t3.user_id
LEFT JOIN rwcdev.user_data_tbl AS t5 ON t5.manager_id = t4.user_id
WHERE t1.user_id = 1
AND    t4.department_id = 7;
```

```
      lev1      |     lev2      |    lev3     |     lev4     |      lev5
----------------+---------------+-------------+--------------+----------------
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Danyelle Eagar
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Erma Luckow
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Jody Baumer
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Max Sundstrom
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Deshawn Fry
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Zane Bossert
 Eugenio Blinn | Gilbert Russi | Kent Hurdle | Bob Eldredge | Al Hultman
```

Output 1 - Simple fixed level query output

This basic approach works to a point when the depth of the hierarchy is known, however the SQL is cumbersome, difficult to optimise and presents problems when hierarchy depth (or number of levels) is not known or can frequently change.

What happens if a level of management is added or removed, or raggedness is introduced (i.e. employees reporting directly into managers or directors).

In order to resolve this problem and make use of some other helpful features, we can use a DBMS in conjunction with a SAS/ACCESS interface.

EXPLOITING DBMS FUNCTIONALITY

Why use a DBMS?

After reading through this paper, some readers will no doubt think this could all be achieved with a combination of SAS Base, SAS Macro and a sprinkling of the SAS SQL Procedure. You might be right, but the end result would most likely be a collection of hard to maintain code which would eventually hit problems (data integrity, performance, and session concurrency) once the application started to scale beyond a small group of end users and data volumes increase.

It is best to use the right tool for the job. In this case that tool is a database management system (DBMS), specifically PostgreSQL.

We can use certain features of the DBMS to make life easier. The initial reasons for selecting PostgreSQL are threefold:

1. It is a well-established, widely supported (free!) open source DBMS with a stable code base and broad support for established SQL standards.
2. SAS utilizes PostgreSQL heavily as part of its product offering, most notably using it for the out of the box SAS Web Infrastructure Platform Data Server. If you have a SAS deployment, chances are you are already running PostgreSQL.
3. The SAS/ACCESS Interface to PostgreSQL provides direct access to DBMS data from within your SAS code, either through implicit or explicit connections allowing you to interact with data within a DBMS as you would with a SAS dataset.

Additionally, the following features of PostgreSQL can be leveraged to deliver an enterprise ready hierarchy management solution:

SEQUENCES

A built in sequence number generator - useful for creating unique IDs for key fields. Can be configured to start at a specific point. Sequences are of data type bigint (8 byte integer) so have a range of -9223372036854775808 to 9223372036854775807. Further options include wrapping/cycling of the sequence to control sequence re-use when the range is exhausted (if your data is sufficiently large) as well as tuneable caching settings to control sequential allocation of IDs in multiuser applications.

DEFAULT VALUES

On creating a table, a column can be defined with a default value, such that when new data is inserted to the table, if a specific column is omitted from the insert statement, then it is automatically assigned the default value. This is extremely useful when used in conjunction with sequences and the nextval function. It can also be used to set timestamps for modification/validity dates automatically by using the database equivalent of SAS's datetime() function.

CONSTRAINTS

Postgres supports many different types of constraints, the most common being check constraints which determine validity for data values in a table. Not-NULL constraints ensure specific values cannot be set to NULL on update/inserts. UNIQUE constraints ensure that data within a column or group of columns is unique for all rows in the table - for example, a user should only have one currently valid user record (unique by user_id and eff_start_date). If a constraint is violated, the action is aborted and an error message is returned to the SAS session.

TRIGGERS

Triggers are events which are fired if certain conditions or rules are satisfied. For example, if a table or column within a table is updated or new data inserted, a trigger can be used to record this event and track the changed data and which user performed the action - i.e. an audit table. Triggers are not covered in this paper, but further information on triggers in postgres can be found in the excellent PostgreSQL documentation[1]

[1] http://www.postgresql.org/docs/9.5/static/sql-createtrigger.html

COMMON TABLE EXPRESSIONS (CTE) - WITH RECURSIVE QUERIES

"A Common Table Expression, or CTE, (in SQL) is a temporary named result set, derived from a simple query and defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement."[2]

The adjacency model can be readily implemented in most modern DBMS's as a result of the implementation of Common Table Expressions (CTE) as outlined in SQL:1999

Should you wish to explore options outside of PostgreSQL, good news because Recursive Common Table Expressions are supported in the following DBMSs:

- IBM DB2 UDB 8 (Dec. 2002)
- Microsoft SQL Server 2005 (Oct. 2005)
- Sybase SQL Anywhere 11 (Aug. 2008)
- Firebird 2.1 (Sep. 2008)
- PostgreSQL 8.4 (Jul. 2009)
- Oracle 11g release 2 (Sep. 2009)
- HSQLDB 2.3 (Jul. 2013)
- Teradata (date and version of support unknown, at least 2009)
- MySQL v8 (2017)[3]

Prior to the implementation of CTE, Oracle has supported recursive queries through the CONNECT BY syntax since version 2 (1977!)

By using a Recursive CTE, a hierarchy of unknown depth can be queried to return a list of subordinates, or in reverse, a list of superiors. This is extremely useful when determining reporting lines, escalation paths and chain of command.

[2] https://en.wikipedia.org/wiki/SQL:1999#Common_table_expressions_and_recursive_queries

[3] https://www.percona.com/blog/2014/02/11/wither-recursive-queries/

Understanding Recursive Common Table Expressions

A recursive CTE typically takes the following form:

```
WITH RECURSIVE <name> (<columns>) AS (
    <initial query>
  UNION ALL
    <recursive query>
)
<select query>
```

Think of the initial query as the base or starting point for the overall query.

The recursive query is then appended to the initial query result via a UNION or UNION ALL statement (normal UNION rules apply here) and is then iterated through until the end of the recursion is detected.

The final select query is used to return items from the generated result set – it can also be used to join to other tables/views in the schema.

Blank space

ORGANIZATIONAL HIERARCHY EXAMPLE

To further demonstrate the adjacency model and the power of the RECURSIVE WITH syntax, the organizational hierarchy will be explored further.

MAINTAINING HISTORY

To facilitate the tracking of changes to the user hierarchy over time, timestamp columns eff_start_date and eff_end_date are used. As changes to user records are made, the eff_end_date column is updated to close the old record off and a new row for that user record (with the updated data) is inserted, with the eff_start_date and eff_end_date columns set to values which determine the record as current/open. This design also allows future or planned changes to be loaded into the database.

VIEWS

We can use an SQL view to return the normalized, current user records. All linked attributes are retrieved via joins and the current date/timestamp (now() function) is used to select the currently valid user records in the user data table using a between clause. Immediate managers are returned in the view for quick retrieval through simple select statements.

The current user view (user_data_current_vw) is defined in the DBMS as follows (only a subset of columns displayed for brevity):

```
CREATE VIEW rwcdev.user_data_current_vw as
SELECT a.user_id,
    a.firstname,
    …
    a.eff_end_date
   FROM rwcdev.user_data_tbl a
     LEFT JOIN rwcdev.role_vw b ON a.role_id = b.role_id
     LEFT JOIN rwcdev.department_tbl c ON a.department_id = c.department_id
     LEFT JOIN rwcdev.location_tbl d ON a.location_id = d.location_id
     LEFT JOIN rwcdev.user_data_tbl e ON a.manager_id = e.user_id
         AND timezone('utc'::text, now()) >= e.eff_start_date
         AND timezone('utc'::text, now()) <= e.eff_end_date
     LEFT JOIN rwcdev.role_vw f ON e.role_id = f.role_id
  WHERE timezone('utc'::text, now()) >= a.eff_start_date
  AND timezone('utc'::text, now()) <= a.eff_end_date
  ORDER BY a.username;
```

Figure 3 shows abridged output from the current user view:

user_id	firstname	lastname	full_name	username	role_id	role_description	role_level	department_id	department	location_id	location
1	Eugenio	Blinn	Eugenio Blinn	EBLINN	2	Managing Director	1	1	Organisation	1	London Old Street
2	Eugene	Strebel	Eugene Strebel	ESTREB	3	Director	2	2	Sales	2	London Bishopsgate
3	Isiah	Delgado	Isiah Delgado	IDELGA	3	Director	2	3	Logistics	3	London Moorgate
4	Len	Levar	Len Levar	LLEVAR	3	Director	2	4	Customer Service	4	Manchester Cross Street
5	Everett	Monterroso	Everett Monterroso	EMONTE	3	Director	2	5	Human Resources	1	London Old Street
6	Columbus	Ricardo	Columbus Ricardo	CRICAR	3	Director	2	6	Accounting & Finance	8	Newcastle
7	Gilbert	Russi	Gilbert Russi	GRUSSI	3	Director	2	7	IT	9	Edinburgh
8	Miles	Bickford	Miles Bickford	MBICKF	4	Manager	5	2	Sales	6	Liverpool

Figure 3 - Abridged output from the current user view

QUERYING SUBORDINATES

When allocating workload or tasks down the chain of command and in turn recording the performance of that piece of work or task, it is a key requirement that it is quick and simple to extract subordinates of a given node/user. By traversing the hierarchy downwards and enumerating the path, the application can be quickly and easily provided with a list of subordinates down all or a selected list of paths.

Let's take a look at how to extract subordinates from a known user in the user hierarchy using the RECURSIVE WITH syntax. Sticking with our example organization, the following query will return Bob Eldredge's (user_id 50) subordinates

```
/* The RECURSIVE WITH syntax is native to the DBMS and must be executed inside an
explicit connection to the DBMS and the result set returned to SAS */
/* Set the thisuserid macro variable to the ID of the user we wish to query. In this case
user id = 50 is Bob Eldredge*/
%let thisuserid = 50;
proc sql;
connect to postgres
        (       server=localhost
                user="&pguser."
                password="&pgpwd."
                database="AppsDatabase"
        );
        create table subordinates as select * from connection to postgres
        (
                WITH RECURSIVE subordinates AS
                (
                select
                        a.user_id
                from    rwcdev.user_data_current_vw a
                where
                        user_id = &thisuserid.
                UNION
                select
                        b.user_id
                FROM    (rwcdev.user_data_current_vw b
                        JOIN subordinates t on t.user_id = b.manager_id)
                )
                /* Now use the list of returned user_ids to join to the current user view
                to extract all current data */
                select
                        u.*
                from    subordinates s
                left join rwcdev.user data current_vw u on s.user_id = u.user_id
                /* Exclude self from results */
                where
                        u.user_id != &thisuserid.
        );
disconnect from postgres;
select * from subordinates;
quit;
```

user_id	firstname	lastname	full_name	username	role_id	role_description	role_level	department_id	department	location_id	location	manager_id	manager_username	manager_name
45	Al	Hultman	Al Hultman	AHULTM	6	Employee	20	7	IT	3	London Moorgate	50	BELDRE	Bob Eldredge
52	Danyelle	Eagar	Danyelle Eagar	DEAGAR	6	Employee	20	7	IT	3	London Moorgate	50	BELDRE	Bob Eldredge
47	Deshawn	Fry	Deshawn Fry	DFRY	6	Employee	20	7	IT	3	London Moorgate	50	BELDRE	Bob Eldredge
51	Erma	Luckow	Erma Luckow	ELUCKO	6	Employee	20	7	IT	2	London Bishopsgate	50	BELDRE	Bob Eldredge
49	Jody	Baumer	Jody Baumer	JBAUME	6	Employee	20	7	IT	9	Edinburgh	50	BELDRE	Bob Eldredge
48	Max	Sundstrom	Max Sundstrom	MSUNDS	6	Employee	20	7	IT	8	Newcastle	50	BELDRE	Bob Eldredge
46	Zane	Bossert	Zane Bossert	ZBOSSE	6	Employee	20	7	IT	4	Manchester Cross Street	50	BELDRE	Bob Eldredge

Figure 4 - Subordinates CTE query output for user_id 50 (Bob Eldredge)

QUERYING SUPERIORS

In order for a business to operate efficiently, the order of command and responsibility must be ascertained, especially when orders or tasks require escalation or sign off from more senior resources. Credit, and in some cases punishment, must also be attributed to the correct person, particularly in relation to performance based employment/targets. The following query returns all superiors of Bob Eldredge (user_id 50)

```
/* The RECURSIVE WITH syntax is native to the DBMS and must be executed inside an
explicit connection to the DBMS and the result set returned to SAS */

/* Set the thisuserid macro variable to the ID of the user we wish to query. In this case
user_id = 50 is Bob Eldredge*/
%let thisuserid = 50;
proc sql;
connect to postgres
        (       server=localhost
                user="&pguser."
                password="&pgpwd."
                database="AppsDatabase"
        );
        create table superiors as select * from connection to postgres
        (
                WITH RECURSIVE superiors AS
                (
                select
                        a.user_id,
                        a.manager_id
                from    rwcdev.user_data_cur_vw a
                where
                        user_id = &thisuserid.
                        UNION
                select
                        b.user id,
                        b.manager_id
                FROM    (rwcdev.user data cur vw b
                        JOIN superiors t on t.manager_id = b.user_id)
                )
                /* Now use the list of returned user_ids to join to the current user view
                to extract all current data */
                select
                        u.*
                from    superiors s
                left join rwcdev.user_data_current_vw u on s.user_id = u.user_id
                /* Exclude the user from result set*/
                where
                u.user_id != &thisuserid.
        );
disconnect from postgres;
select * from superiors;
quit;
```

user_id	firstname	lastname	full_name	username	role_id	role_description	role_level	department_id	department	location_id	location	manager_id	manager_username	manager_name
1	Eugenio	Blinn	Eugenio Blinn	EBLINN	2	Managing Director	1	1	Organisation	1	London Old Street	0	ROOT	Top
7	Gilbert	Russi	Gilbert Russi	GRUSSI	3	Director	2	7	IT	9	Edinburgh	1	EBLINN	Eugenio Blinn
13	Kent	Hurdle	Kent Hurdle	KHURDL	4	Manager	5	7	IT	8	Newcastle	7	GRUSSI	Gilbert Russi
0	Top		Top	ROOT	1	Root	0	1	Organisation	1	London Old Street			

Figure 5 - Superiors CTE query output for user_id 50 (Bob Eldredge)

UPDATING THE USER HIERARCHY

Querying the hierarchy from any given location in either direction has been made quick and easy through the use of the recursive CTE. Updating the hierarchy to reflect organisational changes is less straightforward, but can be made easier through the use of pre-defined standardized macros. In order to develop a standardized approach, we must first understand how the update process is required to work to meet the requirements of the solution.

UPDATING USERS

Updates to user records are carried out as follows:

- Find out which users we need to update
- Extract the current record for each user
- The effective end date (eff_end_date) column for each user in the database is updated to the current date and time minus 1 second (or millisecond) to close off the record
- Update the values/attributes for each user to reflect the change(s) in the extracted record. This now becomes the change record
- Each user change record is inserted into the database with the effective start date (eff_start_date) column set to the current timestamp. The effective end date for the new record is set to the “high timestamp” value of 31/12/5999:23:59:59 to set it as an open/currently valid record
- Planned or future changes are carried out in the same way, except the timestamps for effective start and end dates on the affected records are determined by the user selecting the point in time when the change should become effective
- We can use features of PostgreSQL such as default values to automatically set values on inserts to ensure data integrity. Using a the now() function to automatically set the effective from/to fields removes the need to write code to handle this situation

Figure 6 shows the user_data_tbl in its original state, with no change records.

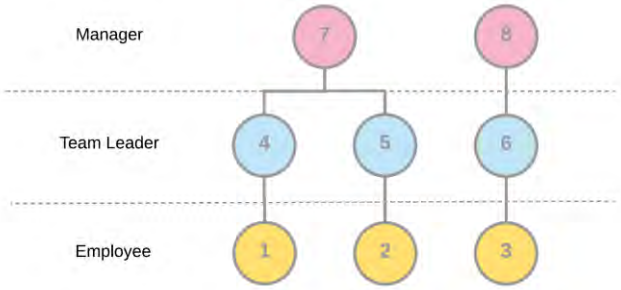
3 levels of the hierarchy from Manager down to Employee are shown.

In this example, 2 employees are changing their manager.

The Change

- User 1 will report into Team Leader 5 with immediate effect
- User 2 will report directly in to Manager 8 to be effective from 09/05/2018

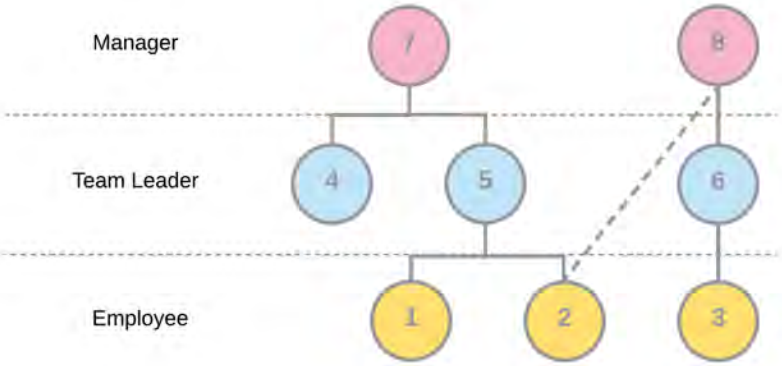
Note: the system date and time of the change is 08/03/2018 @ 11am



user_id	manager_id	eff_start_date	eff_end_date
1	4	01/01/2015:00:00:00	31/12/5999:23:59:59
2	5	01/01/2015:00:00:00	31/12/5999:23:59:59
3	6	01/01/2015:00:00:00	31/12/5999:23:59:59
4	7	01/01/2015:00:00:00	31/12/5999:23:59:59
5	7	01/01/2015:00:00:00	31/12/5999:23:59:59
6	8	01/01/2015:00:00:00	31/12/5999:23:59:59
7	10	01/01/2015:00:00:00	31/12/5999:23:59:59
8	10	01/01/2015:00:00:00	31/12/5999:23:59:59

Figure 6 - Initial user hierarchy and data table

Figure 7 shows the updated user hierarchy and underlying table following the change:



user_id	manager_id	eff_start_date	eff_end_date
1	4	01/01/2015:00:00:00	08/03/2018:10:59:59
2	5	01/01/2015:00:00:00	08/05/2018:23:59:59
3	6	01/01/2015:00:00:00	31/12/5999:23:59:59
4	7	01/01/2015:00:00:00	31/12/5999:23:59:59
5	7	01/01/2015:00:00:00	31/12/5999:23:59:59
6	8	01/01/2015:00:00:00	31/12/5999:23:59:59
7	10	01/01/2015:00:00:00	31/12/5999:23:59:59
8	10	01/01/2015:00:00:00	31/12/5999:23:59:59
1	5	08/03/2018:11:00:00	31/12/5999:23:59:59
2	8	09/05/2018:00:00:00	31/12/5999:23:59:59

Immediate change for user 1 sets prior record to now - 1 second

Future/planned change for user 2 sets the end of the current record to a point in the future

New records for users 1 and 2. User 1 will immediately report into user 5. User 2 will report into user 8 from May 9th 2018

Figure 7 - updated user hierarchy and user data table

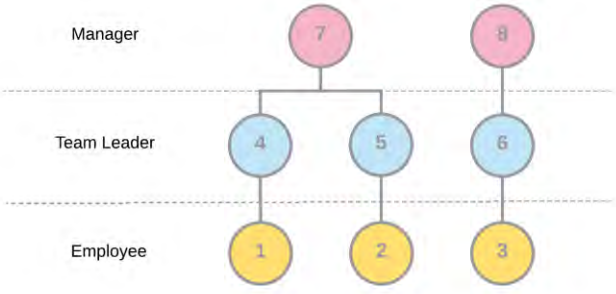
CHANGING MANAGERS

Management changes are carried out using the same methodology as editing individual users.

The Change

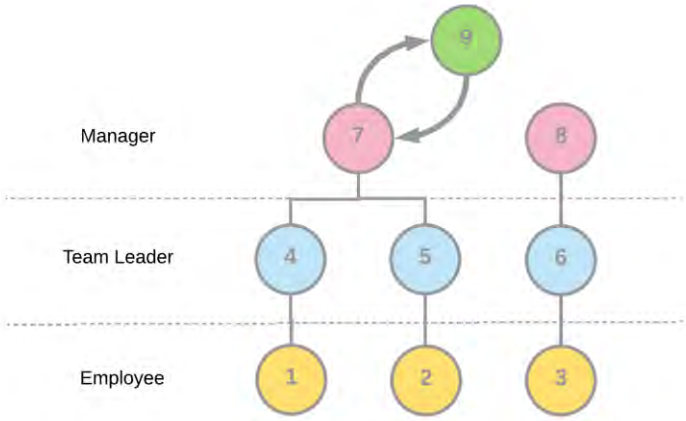
- Manager 7 is to be replaced by a new manager (user_id 9) who has just joined the company.
- All team leaders which report into manager 7 will need to have their user records updated to reflect this change

Note: the system date and time of the change is 08/03/2018 @ 09:31:35



user_id	manager_id	eff_start_date	eff_end_date
1	4	01/01/2015:00:00:00	31/12/5999:23:59:59
2	5	01/01/2015:00:00:00	31/12/5999:23:59:59
3	6	01/01/2015:00:00:00	31/12/5999:23:59:59
4	7	01/01/2015:00:00:00	31/12/5999:23:59:59
5	7	01/01/2015:00:00:00	31/12/5999:23:59:59
6	8	01/01/2015:00:00:00	31/12/5999:23:59:59
7	10	01/01/2015:00:00:00	31/12/5999:23:59:59
8	10	01/01/2015:00:00:00	31/12/5999:23:59:59

Figure 8 shows the change from manager 7 to new manager 9 and the updated and inserted rows in the user data table



user_id	manager_id	eff_start_date	eff_end_date
1	4	01/01/2015:00:00:00	31/12/5999:23:59:59
2	5	01/01/2015:00:00:00	31/12/5999:23:59:59
3	6	01/01/2015:00:00:00	31/12/5999:23:59:59
4	7	01/01/2015:00:00:00	08/03/2018:09:31:34
5	7	01/01/2015:00:00:00	08/03/2018:09:31:34
6	8	01/01/2015:00:00:00	31/12/5999:23:59:59
7	10	01/01/2015:00:00:00	31/12/5999:23:59:59
8	10	01/01/2015:00:00:00	31/12/5999:23:59:59
9	10	08/03/2018:08:13:47	31/12/5999:23:59:59
4	9	08/03/2018:09:31:35	31/12/5999:23:59:59
5	9	08/03/2018:09:31:35	31/12/5999:23:59:59

Close records for Team Leaders 4 and 5 by setting the end of the current record to now - 1 second

New record for new manager/user 9

New valid records for Team Leaders 4 and 5 showing manager and new user 9

Figure 8 - the updated hierarchy and underlying user table after the management change

EXAMPLE USER UPDATE CODE

The example code below is used to update managers using the approach outlined above. Through further use of parameters and macro variables, the below code can be converted into a generic user update macro, to be called with a single line of code – passing the required variables into the macro to update the affected records.

```
*set the system timezone - this must be matched in postgres for current views to work;
*this setting ensures the SAS datetime() function sets the correct timestamp values when updating
records;
options timezone='UTC';
*set the original_mgr_id so we can find the users with this manager;
%let original_mgr_id = 7;
*setup the replacement manager user_id for updating the user records;
%let &replacement_mgr_id = 9;
*Use this to override the effective date of the change if it isn't to be immediate;
%let mgrEffDate = "2008-09-15T15:53:00";

*Create a table of all users with the current manager - only pulling out the currently valid
records from user_data_tbl;
proc sql;
create table update_users_id
as select user_id, eff_start_date
from uhm.user data cur vw where
manager_id = &original_mgr_id.
;

create table update_users as
select a.* from uhm.user_data_tbl a
        inner join update users id b
                on a.user_id = b.user_id
                and a.eff_start_date = b.eff_start_date
where
manager_id = &original_mgr_id.
;
quit;

%macro mgrChange;
        * work out how many users we need to process and put into a variable num_recs;
        data _null_;
                call symput('num_recs',put(numobs,8. -L));
                set work.update_users (obs=0) nobs=numobs;
        run;
        %put &num_recs;
        *start a do loop from the 1st to the nth record;
        %do lcv=1 %to &num_recs;

        *cycle through each record in update_users to be updated and store the user_id we want to
        update;
        data _null_;
                set work.update_users (firstobs=&lcv. obs=&lcv.);
                call symput('user_id',user_id);
        run;

        * delete any existing record with eff_start_date as a future date;
        proc sql;
                delete from uhm.user_data_tbl
                        where user_id eq &user_id.
                        and datepart(eff_start_date) > datetime();
        quit;

        * update and close off current records;
        proc sql;
                update uhm.user_data_tbl
                        set
                        eff_end_date = (datetime() - 1)
                        where user id eq &user_id.
                        and eff_start_date =
                                (select max(eff_start_date)
```

```
                        from uhm.user_data_tbl where user_id = &user_id.);
        quit;

        * extract the most recent record for the user so we can use it for the updated record
        insert;

        proc sql;
                create table updateuser as
                        select *
                        from uhm.user data tbl
                        where user_id eq &user_id.
                        and eff_start_date =
                                (select max(eff_start_date)
                                from uhm.user_data_tbl where user_id = &user_id.);
        quit;

        * update and insert each user record into user table;
        proc sql;
                insert into uhm.user_data_tbl
                        (user id, system key, source system ref, firstname, lastname, username,
                        role_id, department_id, location_id, manager_id, user_active_flg,
                        admin access flg, delg access flg, created_by_id, created_dt,
                        modified_by_id, impersonable_by_id)
                select
                        user id,
                        system_key,
                        source system_ref,
                        firstname,
                        lastname,
                        username,
                        role_id,
                        department_id,
                        location id,
                        &replacement_mgr_id.,
                        user_active_flg,
                        admin_access_flg,
                        delg_access_flg,
                        created_by_id,
                        created dt,
                        &myuserid.,
                        impersonable_by_id
                from updateuser;
        quit;
%end;
%mgrChange;
```

Making additional updates to the code, to create a generic manager change macro, the end result would be called as in the following example:

```
%mgrChange(original_mgr_id=7,replacement_mgr_id=9)
```

PROVING THE SOLUTION AT SCALE

Can the adjacency model/CTE solution handle large data hierarchies? What about very large organizations? Those with staff numbers in the tens or hundreds of thousands. Or those with high labor turnover?

In order to answer this question, I had to find a large hierarchical data source, with many levels in a readily available format to load into a database. Since I'm certainly not the first to tackle this problem, I came across Bill Karwin's presentation[4] showing MySQL 8 CTE functionality. In it he uses the ITIS (the Integrated Taxonomic Information System[5]) which is the *"authoritative taxonomic information on plants, animals, fungi, and microbes of North America and the world"* to prove the performance of the recursive WITH functionality against a large data source.

OBTAINING THE ITIS DATABASE

The ITIS database is available in various formats at the following URL:
https://www.itis.gov/downloads/index.html

The following steps will work on Linux, for Windows please inspect the relevant documentation.

As we're using PostgreSQL in this example, download the PostgreSQL zip file.

```
#download the zip from the ITIS website using wget
wget https://www.itis.gov/downloads/itisPostgreSql.zip

#unzip the downloaded file to a destination directory
unzip ~/itisPostgreSql.zip -d /data/itis

#use psql to load in the psql dump to the desired postgres instance
#The ITIS data will be created in its own database "ITIS" under the public schema
/usr/pgsql-9.4/bin/psql -U testuser -p 5432 -h localhost -f /data/itis/ITIS.sql
```

[4] https://www.slideshare.net/billkarwin/recursive-query-throwdown

[5] https://www.itis.gov/

PREPARING THE ITIS DATA

A pre-built adjacency list table is provided as part of the ITIS data (hierarchy table), however we want to demonstrate the RECURSIVE WITH CTE functionality on the base data table.

The ITIS data includes invalid and rejected taxonomic discoveries in the base table “taxonomic_units” which do not have complete relationships and will therefore break the recursive CTE if included. To achieve a working hierarchy, these invalid records must be removed.

Create a new table “v_taxonomic_units” (valid taxonomic units) with the following CREATE TABLE statement within pgadmin or using psql. You could also use SAS to execute this query if preferred.

```
create table
v_taxonomic_units as
select *
from taxonomic_units
where n_usage in ('valid', 'accepted');


select count(*) from v_taxonomic_units;
count
--------
 565992
(1 row)
```

We now have 565,992 valid taxonomic units arranged in a single table with defined parent-child relationships.

Now create some indexes on the id fields to improve performance of the CTE:

```
CREATE INDEX pki_tsn ON v_taxonomic_units USING btree (tsn);
CREATE INDEX pki_ptsn ON v_taxonomic_units USING btree (parent_tsn);
```

TESTING PERFORMANCE

Test the performance by querying the full taxonomic hierarchy of the species Homo Sapiens.

Homo Sapiens has a taxonomic serial number (tsn) of 180092. Using the same WITH RECURSIVE CTE syntax shown earlier, return the full tree with the following query:

```
-- -- -- -- -- -- -- -- -- -- ---
-- -- Homo Sapiens query -----
-- -- -- -- -- -- -- -- -- -- ---
WITH RECURSIVE superiors AS
(

select
a.tsn,
a.parent_tsn,
a.tsn::text as id_hier_path,
a.complete_name::text as complete_name
from v_taxonomic_units a
-- specify the tsn filter here
where tsn='180092'

UNION

select
b.tsn,
b.parent_tsn,
b.tsn::text ||  '->' || t.id_hier_path  as id_hier_path,
b.complete_name::text as complete_name
FROM (v_taxonomic_units b
              JOIN superiors t on t.parent_tsn = b.tsn)
)

select tsn, parent_tsn, complete_name from superiors;
```

```
  tsn   | parent_tsn | complete_name
--------+------------+---------------
 180092 |     180091 | Homo sapiens
 180091 |     943805 | Homo
 943805 |     180090 | Homininae
 180090 |     943782 | Hominidae
 943782 |     943778 | Hominoidea
 943778 |     943773 | Simiiformes
 943773 |     180089 | Haplorrhini
 180089 |     179925 | Primates
 179925 |     179916 | Eutheria
 179916 |     179913 | Theria
 179913 |     914181 | Mammalia
 914181 |     914179 | Tetrapoda
 914179 |     331030 | Gnathostomata
 331030 |     158852 | Vertebrata
 158852 |     914156 | Chordata
 914156 |     914154 | Deuterostomia
 914154 |     202423 | Bilateria
 202423 |          0 | Animalia
(18 rows)

Time: 1.370 ms
```

The response time for a recursive query against a database of over 500,000 nodes is 1.37 milliseconds. Only the world's largest companies and governmental organizations exceed this number of employees – this level of performance indicates the solution can deliver a responsive foundation on which to build a hierarchy management application.

BUILDING AN APPLICATION

Combining the techniques presented in this paper provides a solid foundation on which to build an application. In its most basic form this could consist of a series of prompt driven stored processes to perform the fundamental tasks required to manage a user hierarchy.

Taking that concept one step further and delivering a fully functioning modern web application is made possible by using the Boemska HTML 5 Data Adapter for SAS (H54S).[6]

The inner workings of the data adapter and how to develop applications using it are beyond the scope of this paper, however more detail can be found in the 2 following excellent papers from previous proceedings:

1372-2017: Build Apps for your Enterprise with SAS ® and HTML5 by Nikola Markovic[7]

1091-2017: Build Lightning Fast Web Apps with HTML5 and SAS® by Allan Bowe[8]

Figure 9 is a screenshot of an example HTML5 User Hierarchy Manager web application as referenced in the abstract, allowing users to manage a full organizational hierarchy through a responsive point and click interface, complete with client side validation.

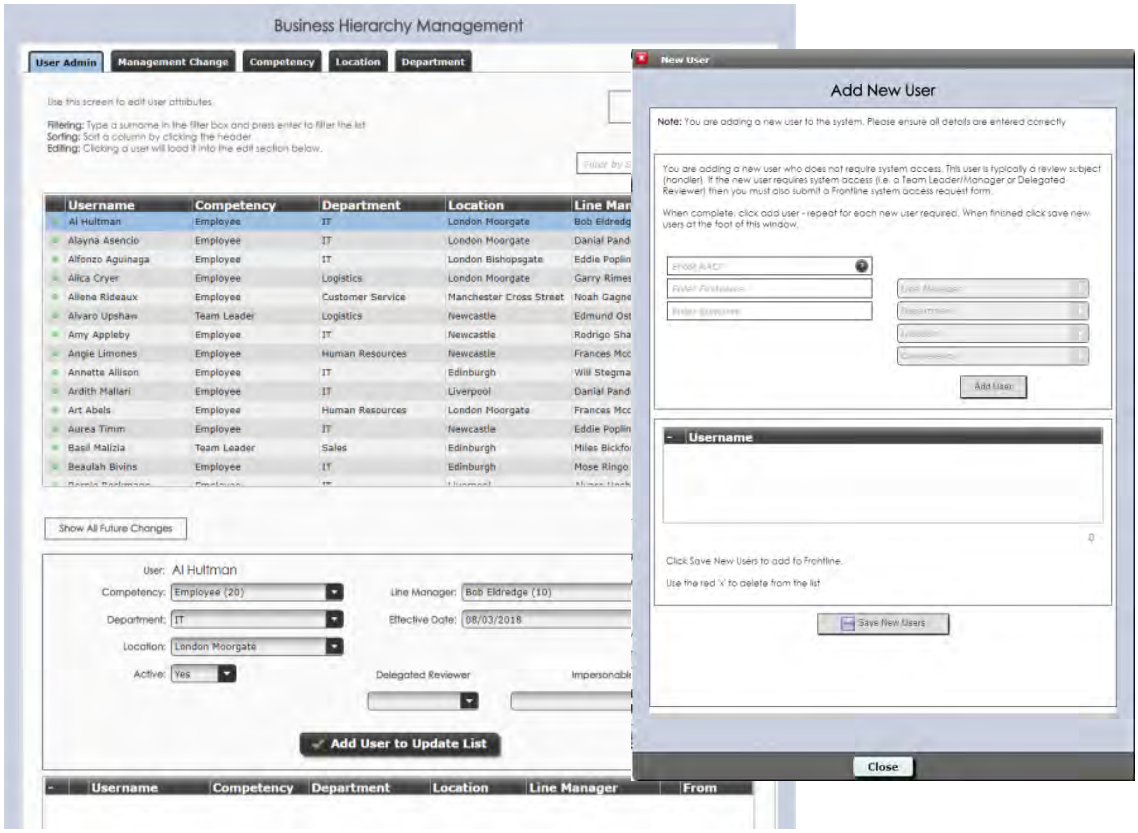


Figure 9 - Example web based user hierarchy management app

[6] https://github.com/Boemska/h54s
[7] https://support.sas.com/resources/papers/proceedings17/1372-2017.pdf
[8] https://support.sas.com/resources/papers/proceedings17/1091-2017.pdf

CONCLUSION

By combining the high performance functionality of PostgreSQL (and other DBMSs) and the unique capabilities of the SAS platform, this paper has presented a standards based solution to a common data problem and provided building blocks for developing an enterprise ready web application on top of the SAS Intelligence Platform

RECOMMENDED READING

- 1372-2017: Build Apps for your Enterprise with SAS ® and HTML5 by Nikola Markovic
- 1091-2017: Build Lightning Fast Web Apps with HTML5 and SAS® by Allan Bowe

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Richard Collins
RWC Technical Solutions LTD
www.linkedin.com/in/richardwcollinsuk
richardwcollins@gmail.com