Paper 1338-2017

Patching the Holes in SQL Systems with SAS®

Christopher P Carson, RTI International

ABSTRACT

As a programmer specializing in tracking systems for research projects, I was recently given the task of implementing a newly developed, web-based tracking system for a complex field study. This tracking system uses a SQL database on the back end to hold a large set of related tables. As I learned about the new system, I found that there were deficiencies to overcome to make the system work on the project. Fortunately, I was able to develop a set of utilities in SAS® to bridge the gaps in the system and to integrate the system with other systems used for field survey administration on the project. The utilities helped do the following: 1) connect schemas and compare cases across subsystems; 2) compare the statuses of cases across multiple tracked processes; 3) generate merge input files to be used to initiate follow-up activities; 4) prepare and launch SQL stored procedures from a running SAS job; and 5) develop complex queries in Microsoft SQL Server Management Studio and making them run in SAS. This paper puts each of these needs into a larger context by describing the need that is not addressed by the tracking system. Then, each program is explained and documented, with comments.

INTRODUCTION

SQL Server is the database of choice for many data management systems. These systems routinely provide a user interface to make it easy for end users to work with the data contained in the SQL tables. In designing a user interface, designers must anticipate how end users will need to access the information and design that functionality. However, user needs can be hard to anticipate.

On a recent project assignment for RTI International, I was given the task of implementing a complex SQL based tracking and control system for a field study that used laptop computers. The project was anything but standard, and I found myself working with a system that was missing some pieces that would be useful and were not implemented in the user interface. There were also situations where the function I needed was addressed, but not in a way I found useful.

The first step in addressing any perceived shortcoming in a SQL system is to develop a general understanding of the underlying table structure. Without some idea of what the main tables are and how they relate to subordinate tables, a developer has little chance of writing queries that add to what the user interface will do “off the shelf.” In addition, a big picture description of what the SQL based tracking system at RTI entails is useful.

BACKGROUND

The SQL based tracking and control system used at RTI International is a custom built in-house application called Nirvana, and the front end developed to run it is called Symphony. RTI is a contract research organization and we serve customers that conduct research projects that may involve numbers of respondents that can range from less than 100 to thousands. In a typical project using the Nirvana tracking system, we load a group of people that are selected as sample members and try to get them to participate in whatever the project entails (often a survey). The ways we try to get their participation and the degree to which they participate are all tracked by defining tracked processes or “stages.” Within each stage are a set of statuses that describe what happened. Examples of processes that are tracked include individual survey participation, mailing out materials, locating sample members, collecting biospecimens, and sending incentive payments.

Tracking and control systems are comparable to Customer Relationship Management software (or CRM). As defined by webopedia, CRM is a category of software that covers a broad set of applications and software designed to help businesses manage customer data and customer interaction, access business information, automate sales, provide marketing and customer support, and also manage employee, vendor and partner relationships.

The Nirvana tracking system has a relational database structure. Figure 1 below shows a high level overview of the main tables underlying this system and their relationships to each other. Two tables are redacted to avoid giving away too many design details. The rest resembles a CRM system.

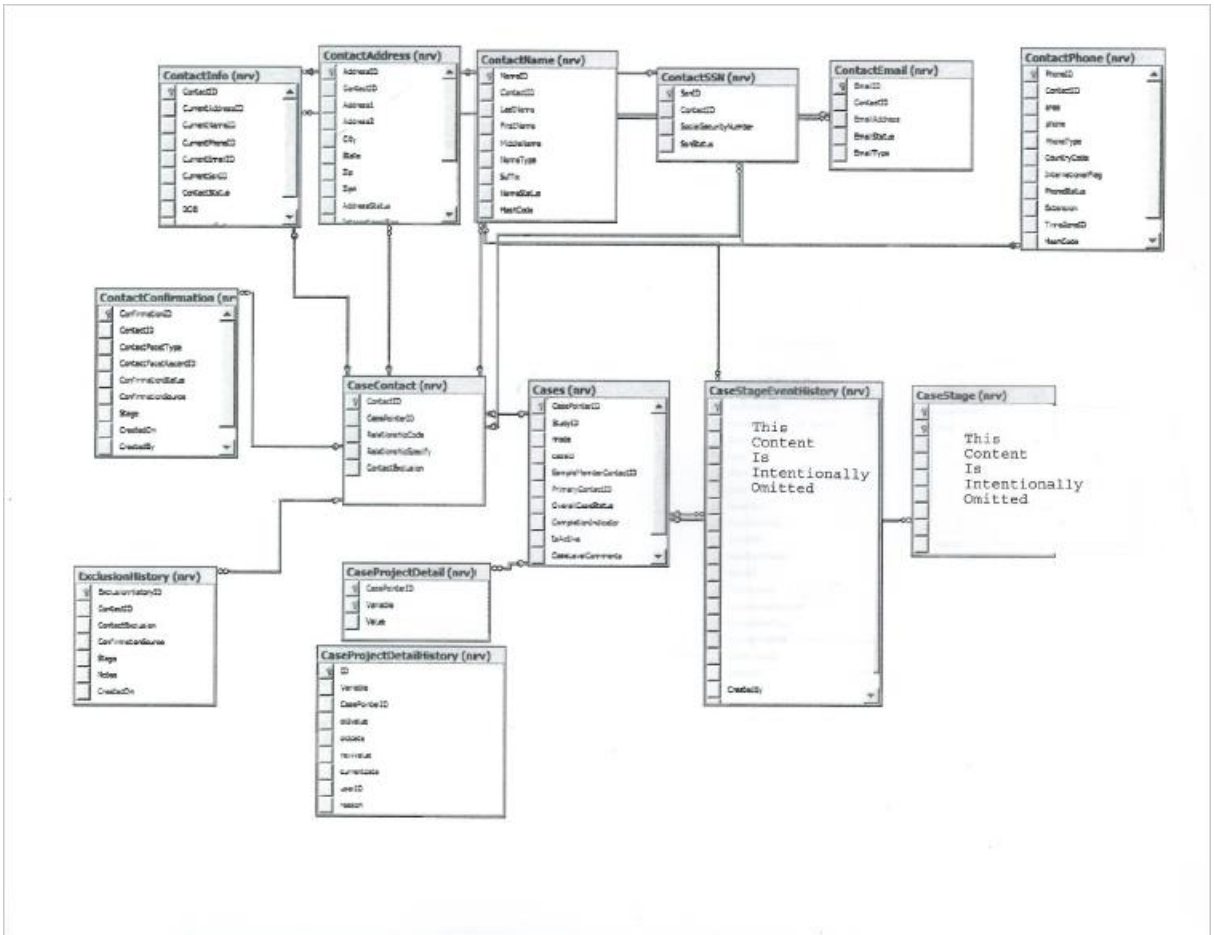


Figure 1 – Tracking and Control System Design Overview (larger view in appendix)

This system also had a fundamental goal of bringing related systems together under one database. Prior to that, systems developed for projects had evolved over time into independent systems with minimal connectivity. Managing projects with multiple modes of data collection was complicated by the lack of connectivity between systems. Now, a SQL database has a separate schema for each sub-system. The schema "fld" (short for field), holds tables that make up the structure of the field system used to support field data collection on laptops. The schema "tops" (short for "tracing operations"), holds tables that underlie the tracing system which is used to find people. The schema "cati" (short for computer aided telephone interview), contains the tables that support RTI's call center. The central schema called "nrv" (short for Nirvana) contains the tables shown in Figure 1. This is the central schema because the other schemas are queried to find and create status updates to be fed into the Nirvana tracking system.

GAPS IN THE USER INTERFACE

The Symphony interface to the Nirvana system has many features that are useful and well designed. However, there are a few places where system enhancements are needed using other tools. SAS has been my choice for providing increased functionality. For example, the user interface provides a flexible way to find all the cases that have a given current status code. It also provides for a way to identify cases with a unique combination of status codes across two tracked processes. But what if I need three or more tracked processes involved in the selection criteria? Or what if I need to know which cases have a given current status but have no status at all in another tracked process? By using SQL within SAS to find them, I can use as many tracked processes as I need in the selection criteria. And by using the in= dataset option, I can easily compare cases across tracked processes and find the cases that are represented in one tracked process but not in another.

I also have the occasional need to set a new status for a list of cases when the list cannot be identified through the user interface at all. In this situation, I have SAS read an external list and perform an update for the cases in the list. SAS provides several ways to read external files, and a little code built on one of them can easily be used to set the status needed.

USING SAS TO BRIDGE THE GAPS

The first step in using SAS with SQL containing multiple schemas is to connect to each SQL schema needed through a libname statement. The following libname statement connects to the nrv schema using a windows “trusted connection.”

```
libname nirvana ODBC complete='DRIVER=SQL Server;
SERVER=RTPWCLV99\PRIVPROD,1433; DATABASE=CHATS; Trusted_Connection=True;'
schema=nrv;
```

Alternatively, the libname statement below does not use a trusted connection but uses a username and password to make the connection.

```
libname nirvana ODBC complete='DRIVER=SQL Server;SERVER=RTPWCLV20;
DATABASE=CHATS; UID=myaccountcredential; PWD=mypassword;' schema=nrv;
```

In these examples, “nirvana” becomes the logical name for the library of tables within the “nrv” schema. To reference the set of tables that make up a different schema, one would only need to name the schema and create a different libname identifier for it. The statement below references the same database, but specifies the “fld” schema within it.

```
libname field ODBC complete='DRIVER=SQL Server;
SERVER=RTPWCLV99\PRIVPROD,1433; DATABASE=CHATS; Trusted_Connection=True;'
schema=fld;
```

These statements unlock the doors to all the tables underlying these systems. The primary way developers use them is to create temporary work tables using Proc SQL and then do further processing using those work tables in data and proc steps. The example program below shows a relatively simple program that reads an Excel table, adds a flag to indicate which cases have a history of undeliverable mail returns, and then writes out a new Excel file to include this new flag:

```
libname nirvana ODBC complete='DRIVER=SQL
Server;SERVER=RTPWCLV99\PRIVPROD,1433; DATABASE=CHATS;
Trusted_Connection=True;' schema=nrv;
libname out1 '\\rtifile02\ProjectShare\RCD\SAS\';
filename in1 '\\rtifile02\ ProjectShare
\Data_Collection\MailMerge\Intro_Letter06162016.xls';
```

```
Proc SQL;

Create table undel as
SELECT distinct Cases.caseid FROM  nirvana.CaseStage INNER JOIN
nirvana.Cases ON CaseStage.CasePointerID = Cases.CasePointerID
WHERE (CaseStage.Stage = 575) AND (CaseStage.Status = 1060) OR
      (CaseStage.Stage = 577) AND (CaseStage.Status = 1060)
order by caseid;
Quit;

proc import
datafile=in1
out=mailfile
dbms=excel
replace;
getnames=yes;
run;

proc sort data=mailfile; by caseid; run;

data mrg;
   merge undel(in=aaa) mailfile(in=bbb);
   by caseid;
   if bbb;
   if aaa then undel='Y';
run;

proc export dbms=excel data=mrg
outfile="\\rtifile02\Wave_V\Data_Collection\MailMerge\Intro_Letter06162016_
modified.xls" replace;
run;
```

The SQL query in this example shows how the control system is being queried to find case IDs that have a recorded history of undeliverable mail statuses. Fortunately, the programmer can pull up the control system user interface and see what statuses get recorded when undeliverable mail is returned and what tracked processes (or “stages”) they get recorded under. These control system details are configurable by the programmer setting up the system. However, design details are often less observable. In a complicated database structure, some of the details one would like to extract are difficult to find. In this next section, I offer suggestions about how to use a combination of tools effectively to develop more complicated queries.

COMBINING TOOLS EFFECTIVELY

Once you have the credentials to programmatically access a SQL database, you should also be able to access it with SQL Server Management Studio. A powerful tool within SQL Server Management Studio is the interactive query builder. For those of us that find these visual tools intuitive, using them to build experimental queries and then copying them over to run in SAS can be vastly more navigable than writing the SQL statement from scratch. The interactive tool can even function as a way to learn how to write SQL statements of higher complexity.

Figure 2 shows a visual representation of a query in the SQL Server Management Studio query builder. This query was written to find out how many cases were updated by a certain user account. The

researchers were interested in cases updated after August 20, 2014, so the temporary table is further refined by defining a cutoff date and eliminating records dated earlier than the cutoff.

The image shows the resulting query as the design was built up visually. The designer does not need to know exactly what all the relationships are between tables, because the SQL database has all the relationships permanently defined. SQL database relationships are permanently defined to enforce "referential integrity." Referential integrity ensures that any foreign key field must agree with the primary key that is referenced by the foreign key. This allows a new table to be dropped into the design and SQL automatically places the relationship lines that indicate how the table joins are specified. SELECT queries will not damage or alter underlying data in tables, so experimenting with unfamiliar designs or tables is encouraged. If a table looks like it may be related and may have data items of interest, drop it into the design and see if the visual tool shows relationships with other tables.

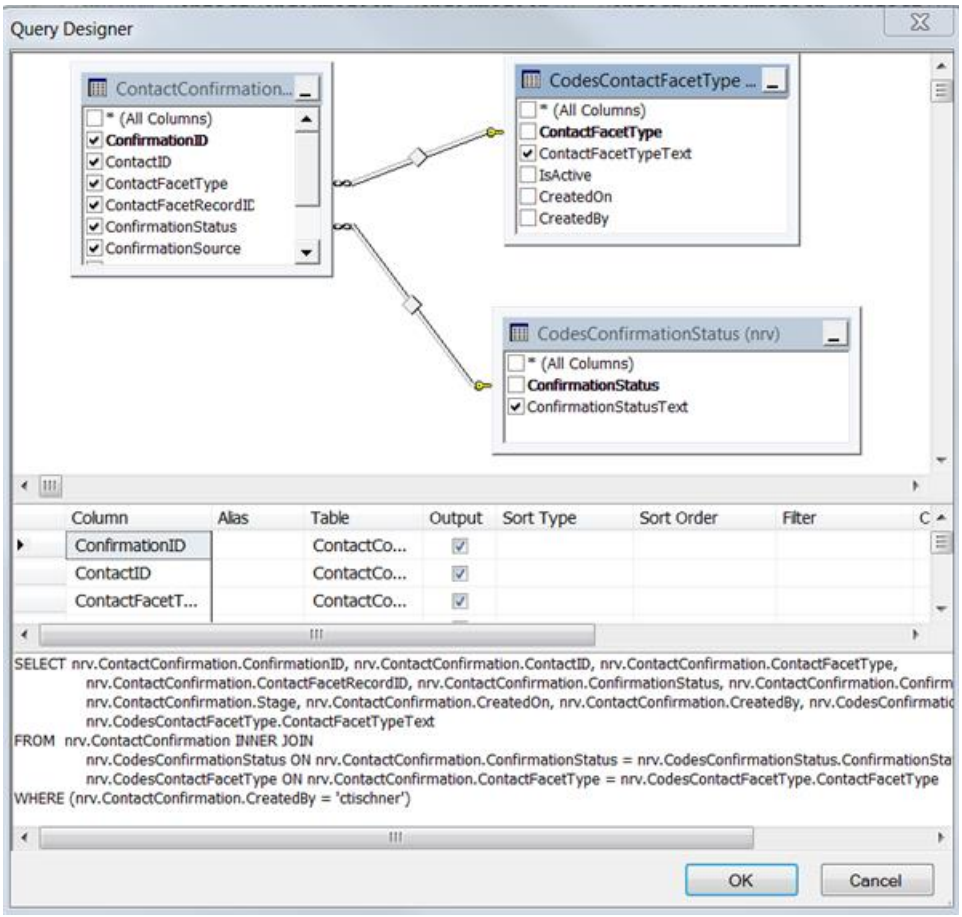


Figure 2 – SQL Server Interactive Design Tool

Once you are satisfied that a new query design would be useful to include in a SAS program under development, you can easily adjust it to make it run within Proc SQL. When copying a query from the SQL Server Management Studio query builder, remove all the references to the schema when that reference is in a field definition. If it is in a table definition, replace it with the logical defined in the libname statement. For example, the statement shown in Figure 2 would be modified as follows to run in SAS Proc SQL.

```
SELECT nrv.ContactConfirmation.ConfirmationID,
nrv.ContactConfirmation.ContactID,
nrv.ContactConfirmation.ContactFacetType,
                nrv.ContactConfirmation.ContactFacetRecordID,
nrv.ContactConfirmation.ConfirmationStatus,
nrv.ContactConfirmation.ConfirmationSource,
                nrv.ContactConfirmation.Stage,
nrv.ContactConfirmation.CreatedOn, nrv.ContactConfirmation.CreatedBy,
nrv.CodesConfirmationStatus.ConfirmationStatusText,
                nrv.CodesContactFacetType.ContactFacetTypeText
FROM  nrv.ContactConfirmation INNER JOIN
                nrv.CodesConfirmationStatus ON
nrv.ContactConfirmation.ConfirmationStatus =
nrv.CodesConfirmationStatus.ConfirmationStatus INNER JOIN
                nrv.CodesContactFacetType ON
nrv.ContactConfirmation.ContactFacetType =
nrv.CodesContactFacetType.ContactFacetType
WHERE (nrv.ContactConfirmation.CreatedBy = 'cs_user')
```

In this query, all references to “nrv.” in red should be deleted because they are referring to fields within tables. All references to “.nrv” in green should be expanded to the full logical name “nirvana” set up in the libname statement that makes the connection to the SQL database possible.

SQL under SAS is enhanced by the ability to run SAS operators and functions not found in standard SQL. Examples include the two vertical lines in the SAS concatenation operator (||), which can be used in SQL within SAS and is not part of standard SQL. You can also embed SAS functions in SQL statements. One of my favorites is to use the Propcase function to standardize address fields as shown in the incomplete query below.

```
create table tbl_curcontact as
select  c.caseid, propcase(cn.firstname) as firstname, propcase(cn.middlename) as middlename,
propcase(cn.lastname) as lastname,
propcase(ca.address1) as address1,
propcase(ca.address2) as address2,
propcase(ca.city) as city,  ca.state, ca.zip from . . .
```

Running SQL within SAS is also a great way to create run-time macro variables. The following code creates a macro variable called grpcnt by counting the number of observations in a temporary dataset meeting a given condition. The random numbers are then added to a temporary dataset.

```
Proc SQL;
 create table sampgroup as
 SELECT Cases.caseid FROM  nirvana.CaseProjectDetail INNER JOIN
        nirvana.Cases ON CaseProjectDetail.CasePointerID = Cases.CasePointerID
 WHERE (CaseProjectDetail.Variable = 'SampGroup') AND (CaseProjectDetail.Value = '2a');

 Select count(*) into :grpcnt from sampgroup;
quit;

data ids;
  retain seed 985902;
```

```
  do i=1 to &grpcnt;
    ran1 = int(ranuni(seed) * 1000000);
    ran2 = put(ran1, z7.);
    output;
  end;
run;

data nodup(keep=caseid ran2 RVar);
   merge ids sampgroup;
/* no by group is intentional to randomly assign a random # to the case */
run;
```

The next example demonstrates at least two new points. The program processes new interviews through the control system by finding cases that have progressed further through a web interview collection process and posting an updated status for those that have. It also records a date for future reference. To make this code more understandable, it should be noted that the web interview is presented as one module for some sample members (where QDG=1), and as two modules for others (where QDG=2). If a sample member completes the first module but not the second, we want to record the date they started their participation so we can remind them to come back and finish later. In a nutshell, this program does the following:

- Connects to the SQL Database.
- Creates four temporary work tables with Proc SQL.
- Merges data gathered with the password used as the primary key with another table that has the regular case ID.
- Merges related data and executes data step logic to find the next status to be set.
- Constructs statements with string operations to call a stored procedure for each new status change.
- Constructs insert statements with string operations.
- Uses include statements to pull in the statements constructed and executes them. This is considered “pass through SQL” because the execution of these statements is done by SQL Server.

The program is provided below with an enhanced level of comments:

```
libname in2 '\\rtiserver\CHATS\RCD\BatchJob\CodeBookS1\Data\';
libname in1 '\\rtiserver\CHATS\RCD\SAS\';

libname nirvana ODBC complete='DRIVER=SQL
Server;SERVER=RTPWCLV99\PRIVPROD,1433; DATABASE=CHATS;
UID=my_useraccount;PWD=my_password;' schema=nrv;
filename out2 '\\rtiserver\CHATS\RCD\SAS\Status_Updates.txt'; /* file for
status updates */
filename out3 '\\rtiserver\CHATS\RCD\SAS\ModACompdate_Updates.txt'; /* file
for recording the date something happened */

proc SQL;
```

```
/* create a table of passwords that might be found in newly completed
interviews */
create table PASS as
SELECT CaseProjectDetail.CasePointerID, CaseProjectDetail.Variable,
CaseProjectDetail.Value as PW, Cases.caseid FROM  nirvana.CaseProjectDetail
INNER JOIN nirvana.Cases ON CaseProjectDetail.CasePointerID =
Cases.CasePointerID
WHERE (CaseProjectDetail.Variable = 'Password') and (cases.mode = 4)
order by PW   ;

/* create a table showing what Questionnaire Design Group each sample
member is in */
   create table QDG as
   SELECT CaseProjectDetail.Value as QDG, Cases.caseid
   FROM  nirvana.CaseProjectDetail INNER JOIN
   nirvana.Cases ON CaseProjectDetail.CasePointerID = Cases.CasePointerID
   WHERE (CaseProjectDetail.Variable = 'QDG') order by caseid  ;

   /* create a table of previously recorded dates that sample members did
module A of the interview */
   create table AComp as
   SELECT CaseProjectDetail.Value as ACompDate, Cases.caseid
   FROM  nirvana.CaseProjectDetail INNER JOIN
   nirvana.Cases ON CaseProjectDetail.CasePointerID = Cases.CasePointerID
   WHERE (CaseProjectDetail.Variable = 'ACompDate') order by caseid  ;

   /* create a table showing the current status of each case */
   create table curstat as
   SELECT CaseStage.Status as currentstatus, Cases.caseid
   FROM  nirvana.CaseStage INNER JOIN
   nirvana.Cases ON CaseStage.CasePointerID = Cases.CasePointerID
   WHERE (CaseStage.Stage = 700) order by caseid ;

quit;

/* For these web interviews, the password is in the caseid field */
/* The real caseID will be picked up in the new merge */
data in1.webcases;
   set in2.wavev_raw(rename=(caseid=pw));
run;

proc sort data=in1.webcases; by pw; run;
proc sort data=PASS; by pw; run;

data interviews; /* this picks up the caseid and casepointerid */
   merge PASS(in=bbb) in1.webcases(in=aaa);
   by PW;
   if aaa and bbb;
run;

/* now get ready to merge on real caseID */
proc sort data=interviews;  by caseid; run;

data interviews2;
   merge interviews(in=ccc) QDG(in=ddd) curstat AComp;
   by caseid;
   if ccc and ddd;
```

```
   length cp $ 3;

   cp = put(casepointerid, $5.);

/* Now write a status update record */
/* The status a case qualifies for depends on the value of QDG */
/* Values found in fields within the interview record dictate what status
update a case should get. */
   file out2;

   if QDG = 1 then do;
    if (ModuleA_Complete = 'Y' and ModuleB_Complete = 'Y') or LOQ 'END'
    then Status = '2693';
    else if ModuleA_Begin = 'Y' then Status = '1247';
   end;
   if QDG = 2 then do;
    if ModuleA_Complete = ' ' and ModuleA_Begin = 'Y' then Status = '1245';
    if ModuleB_Complete = ' ' and ModuleB_Begin = 'Y' then Status = '1246';
/* put 'Partial B-' caseid; */
    if ModuleA_Complete = 'Y' then Status = '2691';
    if ModuleB_Complete = 'Y' or LOQ = 'END' then Status = '2692';
/* implies A is completed too */
   end;

   CreatedOn = put(today(), MMDDYY10.);
   EventDate = CreatedOn;

   EventComment = 'Status created with InterviewProcess.sas';
   User = 'CPC';

   Stage = '700';

   /*Put out line to require a status change if the case has progressed */
   if Status gt currentstatus then do;
    /* Use string operations to build up the format of a call to an
    existing stored procedure */
      outstr='Execute (nrv.InsertCaseStageEvent null, ' || CasePointerID ||
', ' || stage || ', ' || Status || ', null, null, ''' || EventComment ||
''', ''' ||
      EventDate || ''', null, null, null, null, null, null, ''' ||
CreatedOn || ''', ''' || User || ''', null) by ODBC;';
      put @1 outstr;
   end;

   /* need to record the date module A was completed */
   if currentstatus ne status and status = 2691 and ACompDate eq ' ' then
do;
      CreatedOn = put(today(), MMDDYY10.);
      ProjectVar = 'ACompDate';
      file out3;
      outstr = 'Insert into nirvana.caseprojectdetail (Casepointerid,
variable, value) values (' || casepointerid || ', ''' || ProjectVar ||
''','''' || CreatedOn || ''');';
      put outstr;
   end;
run;
```

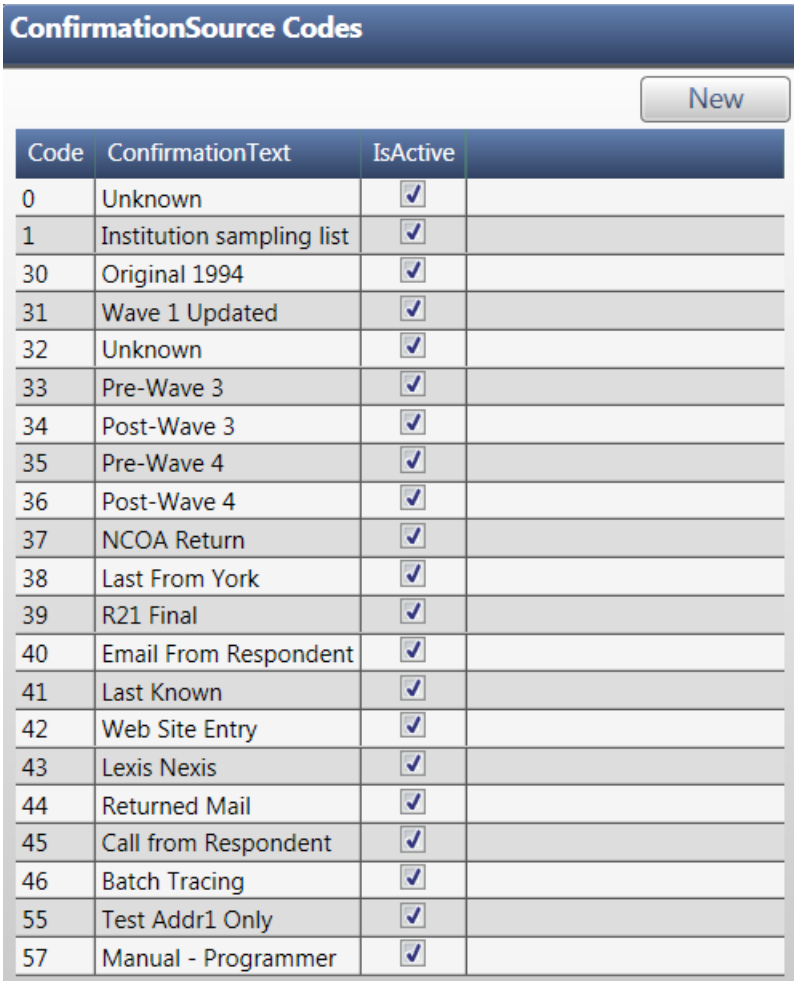
```
proc SQL;
connect to ODBC (complete='DRIVER=SQL
Server;SERVER=RTPWCLV99\PRIVPROD,1433; DATABASE=CHATS;
UID=my_useraccount;PWD=my_password;');

/* a line of ModACompdate_Updates.txt looks like this */
/* Insert into nirvana.caseprojectdetail (Casepointerid, variable, value)
values ( 4856, 'ACompDate','12/29/2016');  */

%include '\\rtifile02\CHATS\RCD\SAS\ModACompdate_Updates.txt';

/* a line of Status_Updates.txt looks like this */
/* Execute (nrv.InsertCaseStageEvent null, 3368, 700, 1247, null, null,
'Status created with InterviewProcess.sas', '12/29/2016', null, null, null,
null, null, null, '12/29/2016', 'CPC', null) by ODBC; /*

%include '\\rtifile02\CHATS\RCD\SAS\Status_Updates.txt';
quit;
```

The next example is a bit unusual. I want to illustrate how a relatively complicated query can be built and used when only limited information is known about where the information is in the underlying tables and how those tables relate to each other.

You can find clues to what is available in the underlying tables of a SQL based system by paying careful attention to what is visible in the user interface. For example, I was recently given the unusual request to provide an address file containing the best address we had at the start of the project, not the last and best address for the respondent. I knew that address history was in the system because I could see them by calling up individual cases. Figure 3 shows how the source of address information is tagged and displayed within the user interface.

Figure 3 – Address Display

I also found a list labeled “confirmation source codes” within the user interface where I could add definitions indicating where loaded addresses had come from. By referring to this list, I found that I wanted to extract the addresses associated with code 41 labeled “last known”.



ConfirmationSource Codes

New

Code	ConfirmationText	IsActive
0	Unknown	☑
1	Institution sampling list	☑
30	Original 1994	☑
31	Wave 1 Updated	☑
32	Unknown	☑
33	Pre-Wave 3	☑
34	Post-Wave 3	☑
35	Pre-Wave 4	☑
36	Post-Wave 4	☑
37	NCOA Return	☑
38	Last From York	☑
39	R21 Final	☑
40	Email From Respondent	☑
41	Last Known	☑
42	Web Site Entry	☑
43	Lexis Nexis	☑
44	Returned Mail	☑
45	Call from Respondent	☑
46	Batch Tracing	☑
55	Test Addr1 Only	☑
57	Manual - Programmer	☑

Figure 4 – Configurable List of Information Confirmation Sources

With these clues, I looked at the list of underlying tables in SQL Server Management Studio and found one called nrv.CodesConfirmationSource. This table gave me what I needed to start building a new query to extract the information needed. I also knew from the structural design summary that the table "ContactInfo" would have to be involved in any query accessing contact information. With this knowledge, I could start to build up a query using the visual query design tool in SQL Server Management Studio.

I knew that finding the tables that might be involved in my query and dropping them into the design tool was a major milestone. Since SQL remembers how the tables relate to each other and will place the relationship lines between the tables for you; you do not have to have prior knowledge of what these relationships are. Appendix A includes screen shots showing the step by step buildout of the query needed to extract the requested information. While the request is specific to the SQL system in use at my workplace, it can be generalized for the larger audience by the following steps.

- Relate the request to what can be seen within the user interface.
- Note what pieces of information are seen in the user interface and what they are called.
- Open the SQL Server database in SQL Server Management Studio and browse the underlying tables.

- Run a SELECT query if a table appears to have information you need.
- Drop tables of interest into the query design tool and observe how the relationship lines are drawn.
- Check off the fields you want returned.
- Add filters or other criteria to further refine the query design.
- Run the query and spot check the results.
- Copy the query and paste it into Proc SQL.
- Adjust the query to take advantage of any SAS functions that may be helpful and to adjust field and file references as previously described.

CONCLUSION

By combining the power of SQL Server Management Studio, Proc SQL, and basic SAS programming techniques, a SQL based system can be extended to address project needs that may come up. The techniques in this paper have been used to implement advanced control systems on several large projects. They are also routinely used for reporting and system integration.

APPENDIX

The following images are the visual buildout of a query that attempted to pull historical address information out of a complex SQL system. The researchers requested address information that was known at the start of a long project.

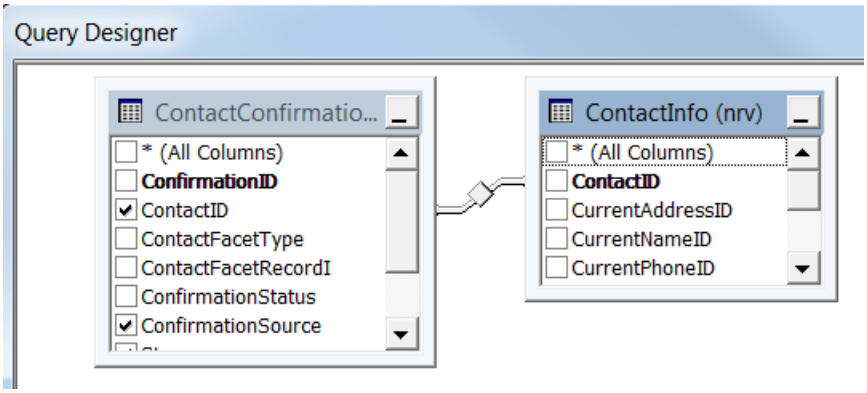


Figure 5 – Building Up a Query Design #1

Next, I dropped in the address table. Note that another join relationship appears with the addition of this new table.

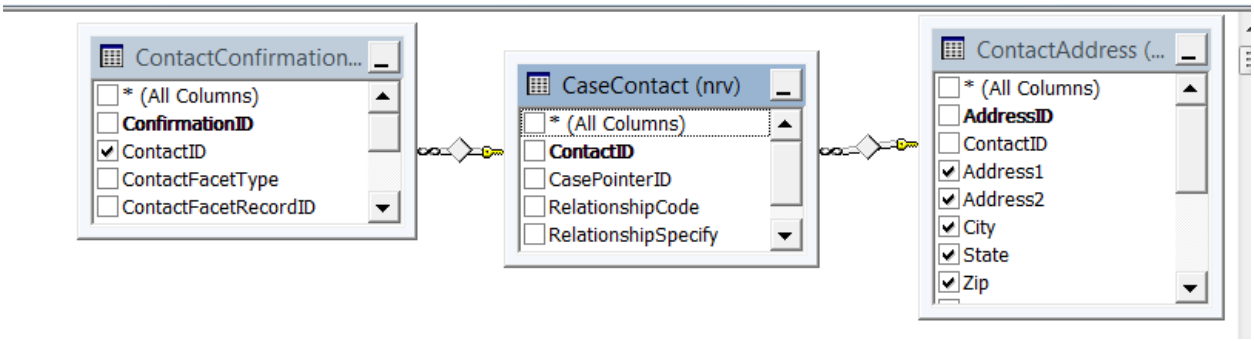


Figure 6 - Building Up a Query Design #2 with Address Table

Finally, I added the table containing the real project ID for every case cases “Cases”.

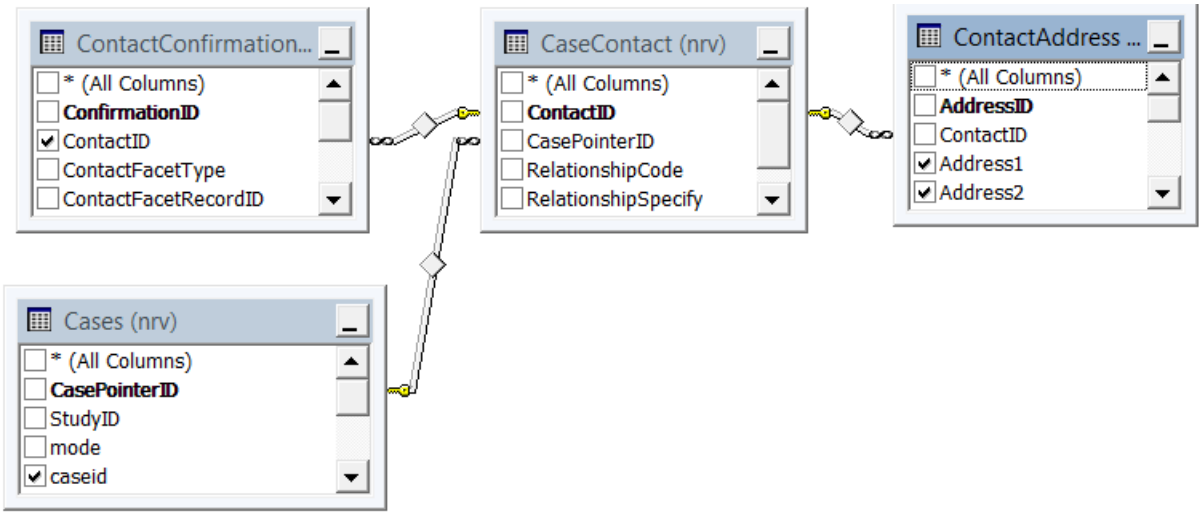
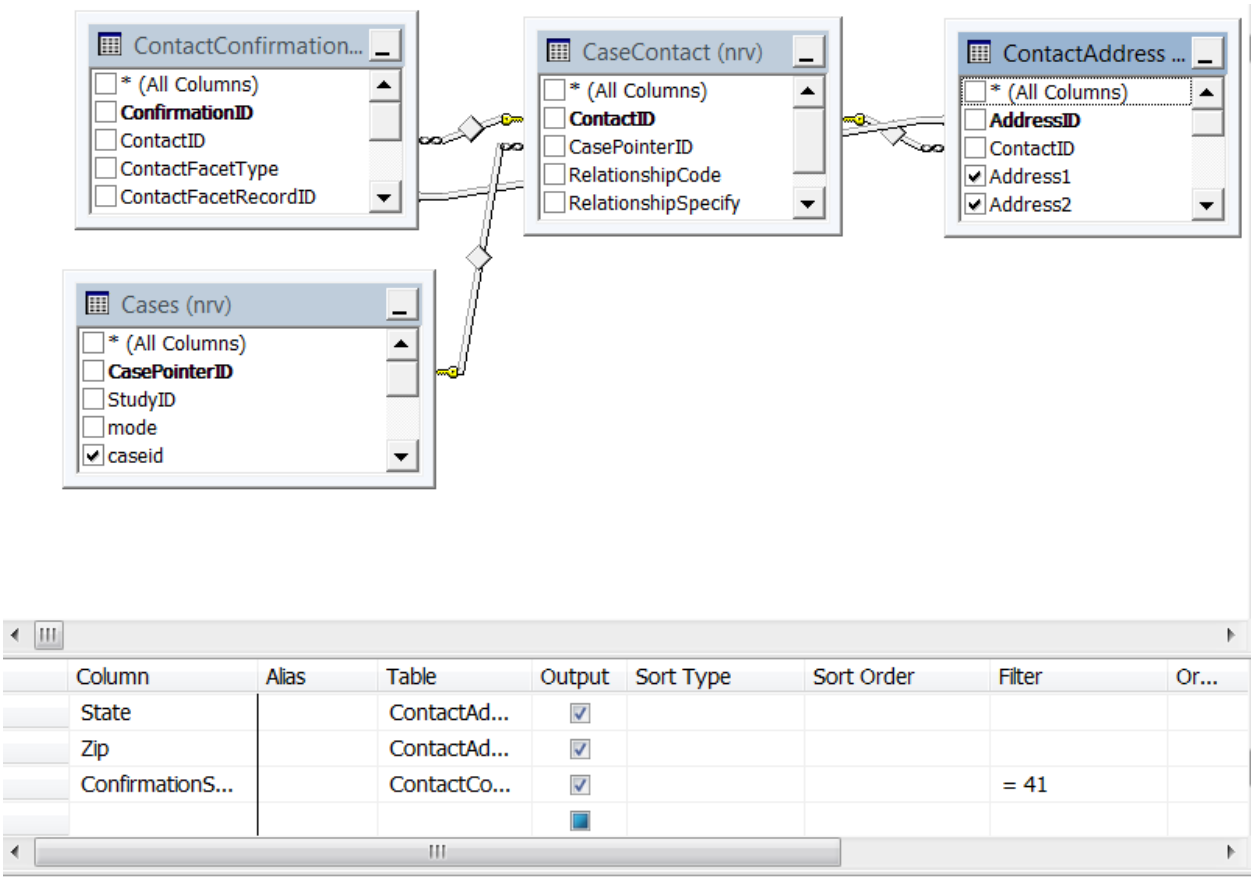


Figure 7 - Building Up a Query Design #3 adding the case ID

At this point in the query build, the result I was getting included all addresses for each sample member. To complete the design, I had to determine how the entries in “ContactConfirmation” related to the “ContactAddress” table. By looking at the values in common between these two tables, I was able to add the last reference needed by linking “ConfirmationFacetRecordID” and “AddressID”. This was the only link not created for me because it was not a permanently saved relationship. Figure #8 below shows the entire visual design, including the addition of a filter value to get the specific confirmation source I was looking for.



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
State		ContactAd...	☑				
Zip		ContactAd...	☑				
ConfirmationS...		ContactCo...	☑			= 41	
			■				

```
SELECT nrv.Cases.caseid, nrv.ContactAddress.Address1, nrv.ContactAddress.Address2, nrv.ContactAddress.City, nrv.ContactAddress.State,
          nrv.ContactConfirmation.ConfirmationSource
FROM  nrv.ContactConfirmation INNER JOIN
          nrv.CaseContact ON nrv.ContactConfirmation.ContactID = nrv.CaseContact.ContactID INNER JOIN
          nrv.ContactAddress ON nrv.CaseContact.ContactID = nrv.ContactAddress.ContactID AND
          nrv.ContactConfirmation.ContactFacetRecordID = nrv.ContactAddress.AddressID INNER JOIN
          nrv.Cases ON nrv.CaseContact.CasePointerID = nrv.Cases.CasePointerID
WHERE (nrv.ContactConfirmation.ConfirmationSource = 41)
```

Figure 8 - Building Up a Query Design #4 with the last relationship needed and a record filter

From this point, the query was ready to copy into my SAS program. The only adjustments needed to make it run within SAS were the “nrv” references previously mentioned.

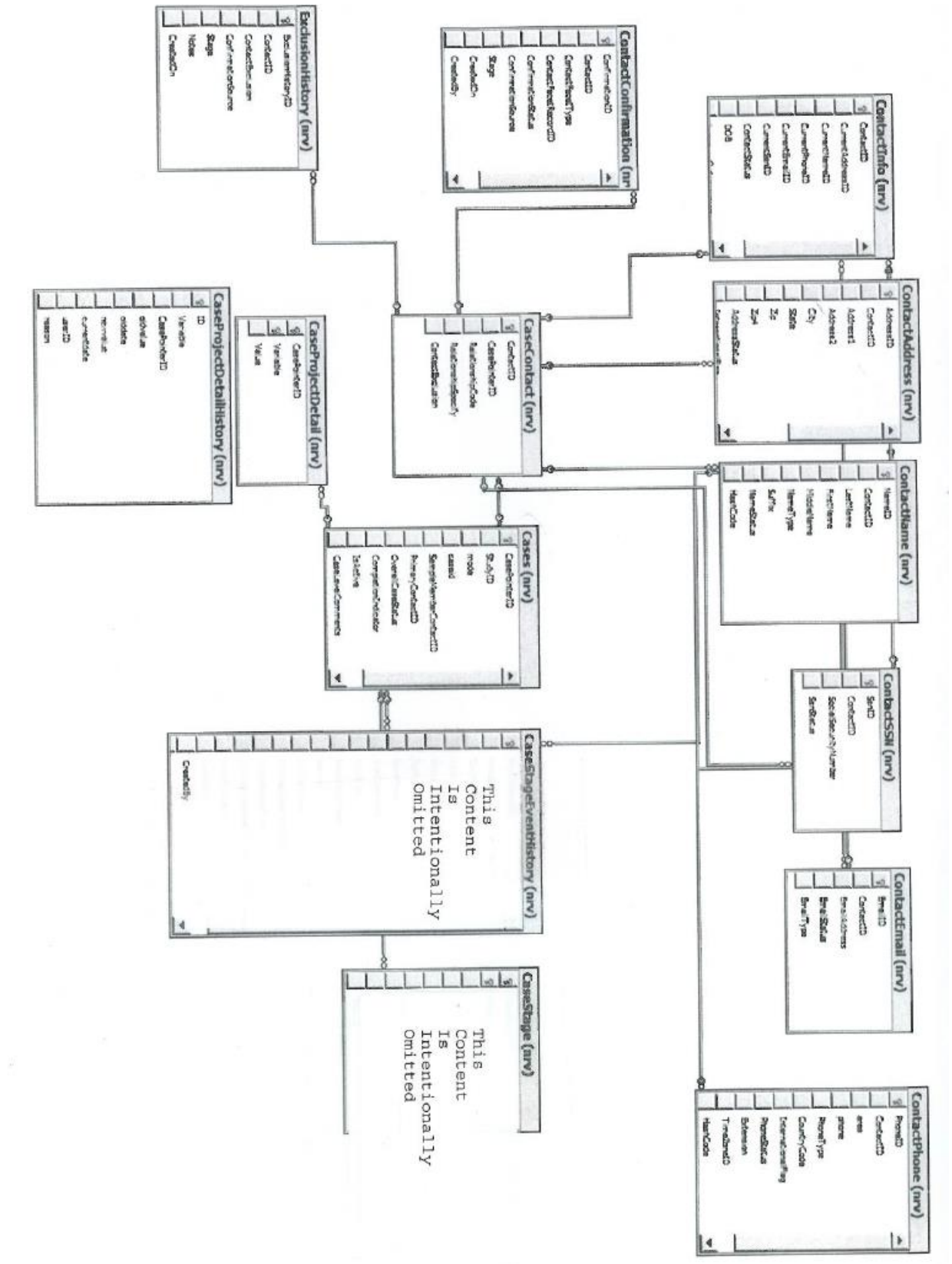


Figure 1 – Tracking and Control System Design Overview (larger view)