Simeon Talmage Baker

# Panko

## Summary

Panko started as an offshoot of CourseChew. I wanted a simple way to be able to audit data and do true-at-the-time reporting, that ended up taking up a lot of my time and has now emerged as an independent tool.

This started as a Temporal Database but having revisited it 6 months later, the idea of a framework to lump on top SQL Server is apparently impractical, though it didn't appear so at the time. Instead I'm condensing the work I've done into a single procedure to be executed on specific tables in order to achieve Point-In-Time reporting on them.

The following document is an overview of the system as it works at the moment. I am hoping that this in depth review will make transforming Panko into a single product easier.

Notes/Thoughts:

- Source/Target table should be explicit nomenclature
- Naming convention generally needs revisiting
- Configuration table
    - Schema names should be variables
    - Add debug variable for print statements
- Create clustered index on delta product table ActiveTo,ActiveFrom,State
- Handle table alter statements
    - Move existing states into hold, create new state table (including mod/removed columns at the end with mod#/del#_ prefix if they were populated), drop and recreate view omitting deleted columns, insert change note.
- Stop beginning statements with terminator ';' just be more careful with how statements are structured, if you miss one it will just fail
- "Manifest" is a bad name for the frist procedure – it should probably be something like "CreatePankos", and perhaps "Panko" could be "Audit"
- Maybe there should be a table in the tsc schema that keeps track of the audited tables.
- ManifestPanko should be the procedure that installs panko as opposed to a database trigger that monitors the input schema

## Components

Simeon Talmage Baker

- Schemas
    - tsc
    - state
    - delta
    - note
    - *Input*
    - *Hold_state*
    - *Hold_change*
- Scalar-value Functions
    - GetABColumnComparison
    - GetColumnCreate
    - GetColumnSelect
    - GetFirstUniqueColumn
    - NumberOfColumns
- Stored procedures
    - CreateUniqueColumn
    - CreatePankoTables
    - HoldPanko
    - RebuildPankoTriggers
    - CreatePankoTriggers

Simeon Talmage Baker

- DDL Triggers:
    - ManifestPanko
    - HoldPanko

## Schemas

I've chosen to use separate schemas in this project to be able to keep object names universally the same- this will help prevent name clashing and makes it easier to write the dynamically generated SQL that will power the system. Here are the schemas I've chosen and the role I expect them to fulfil

### tsc

Standing for Table State Change, I'm using this schema instead of *dbo* to separate the functional database objects of this system from existing procedures so they can be implemented without difficulty and managed separately from existing database system developments.

### state

The *state* schema stores the unique iterations of the complete set of rows for it's given table. States are being used to minimise data redundancy.

### delta

The *delta* schema (formerly *change*) is the functional aspect of Panko, tables stored in *delta* will always have the exact same columns: ChangeId, State, Step, ActiveFrom, ActiveTo, Author. I've decided to use the name *delta* because:

- It makes clear distinctions in language when talking about the action of change as opposed to database objects (the schema or table)
- It will show up after "*dbo*" in the object explorer
- It's the same number of characters as "*state*"

### note

This schema will house a table to detail what changes have been made, I thought it was important to keep this separate from the *delta* schema because the details of changes could be quite long and impact performance

### input

The *input* schema is what the system currently (18-Apr-2020 17:40) used to identify tables that should be audited. This will be removed as Panko becomes targeted on specific tables within existing systems as opposed to using a catch-all schema

### hold_x

The *hold* schemas are used to preserve dropped table data. This has pros and cons as well as technical issues.

- Tables may be dropped for capacity reasons (though this could be resolved by dropping the hold tables if done with appropriate privileges and system knowledge)
- If a table is dropped and then recreated with different columns the *state* table will not be compatible and the triggers on these tables will fail

I'll keep the idea hold for now and consider its utility later- at the moment the system is cascading the drop to *state* and *delta* objects.

## Scalar-value Functions

The functions in this project generally return information about tables, used for building or updating the product tables. No functions are co-dependant/inter-related as of yet and are all used primarily in stored procedures

## GetABColumnComparison

This function generates a column comparison between two tables using the same set of columns to check equivalency. This builds an INTERSECT (or NOT EXCEPT) function in the WHERE clause of dynamic SQL without need for a CTE. It is used instead of EXCEPT because it allows you to excude the “StateId” which is useful for being returned in the select and isn’t included in the target table being audited.

This function is used once at the moment, in the CreatePankoTriggers procedure. It is used to ensure that duplicate states are not entered and to retrieve existing *StateId*’s for the *[delta]* table.

```
CREATE FUNCTION [tsc].[GetABColumnComparison] (
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
)
RETURNS varchar(max)
AS BEGIN
        declare @ColumnCompareList varchar(max)

        SELECT @ColumnCompareList = '('
        + STUFF((
                SELECT ' and ( A.['+ c.name +'] = B.['+ c.name +'] or ( A.['+ c.name +'] is null and B.['+
c.name +'] is null ) )'

                FROM sys.tables                  t
                INNER JOIN sys.columns   c on c.object_id = t.object_id

                WHERE schema_id = ( SELECT schema_id FROM sys.schemas WHERE name = @schemaname)
                and t.name = @tablename

                FOR XML PATH(''),type).value('.','varchar(max)')
        ,1,5,'') + ')'

        RETURN(@ColumnCompareList)
END
```

## GetColumnCreate

This function returns a comma separated list of columns in a table as well as their type definition. It is used to generate the State table and purposefully ignores the NULLABLE variable and other constraints to account for column alterations.

This function is used once at the moment, in the CreatePankoTables procedure to generate the *[state]* table

```
CREATE FUNCTION [tsc].[GetColumnCreate] (
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
)
RETURNS varchar(max)
AS BEGIN
        declare @ColumnCreateList varchar(max)

        SELECT @ColumnCreateList =
        STUFF((
                SELECT ',[' + c.name + '] ' + t.[name] + CASE WHEN c.[precision] = 0 THEN
'('+isnull(nullif(cast(c.max_length as varchar),'-1'),'max')+')' ELSE '' END + ' NULL'
                FROM            sys.tables                d
                INNER JOIN      sys.schemas               s on d.schema_id = s.schema_id
                INNER JOIN      sys.columns               c on d.object_id = c.object_id
                INNER JOIN      sys.types                 t on c.system_type_id = t.system_type_id

                WHERE s.[name] = @schemaname
                and d.[name] = @tablename
                and LEFT(t.name,3) <> 'sys'
                ORDER BY column_id
                FOR XML PATH(''),type).value('.','varchar(max)')
        ,1,1,'')

        RETURN(@ColumnCreateList)
END
```

## GetColumnSelect

This function is very similar to the GetColumnCreate function in that it returns a comma separated list of columns in a table but without the column data type.

It is used for two components in the CreatePankoTriggers procedure

1. In order to define @ColumnListUniqueFirst, this variable is necessary to align state columns with deletion inserts performed by the tscDeleteState_ product trigger
2. To INTERSECT/EXCEPT existing/new states when a target table is updated in the tscUpdate_ product trigger

```
CREATE FUNCTION [tsc].[GetColumnSelect] (
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
        ,@tablealias as varchar(128) = null
)
RETURNS varchar(max)
AS BEGIN
        declare @ColumnSelectList varchar(max)

        SELECT @ColumnSelectList =
        STUFF((
                SELECT ',' + isnull(@tablealias+'.','') + '[' + c.name + ']'
                FROM            sys.tables                  d
                INNER JOIN      sys.schemas                 s on d.schema_id = s.schema_id
                INNER JOIN      sys.columns                 c on d.object_id = c.object_id
                WHERE s.[name] = @schemaname
                and d.[name] = @tablename
                ORDER BY column_id
                FOR XML PATH(''),type).value('.','varchar(max)')
        ,1,1,'')

        RETURN(@ColumnSelectList)
END
```

## NumberOfColumns

This function returns the number of columns in a given table or view as a smallint.

This is used in the CreatePankoTriggers procedure to create a row of null values where a row has been deleted in the tscDeleteState product trigger

```
CREATE FUNCTION [tsc].[NumberOfColumns] (
         @schemaname as varchar(128)
        ,@objectname as varchar(128) --Or View
)
RETURNS smallint
AS BEGIN
        RETURN ( SELECT count(name) FROM sys.columns WHERE columns.object_id = OBJECT_ID(@schemaname + '.' +
@objectname) )
END
```

## GetFirstUniqueColumn

This function will return the first unique column in the following order of importance: Primary key; Identity column; The first column with a unique constraint.

It is used in the CreatePankoTriggers procedure to define the @UniqueColumn variable used across all product trigger logic.

It is also used in the CreateUniqueColumn procedure to check a unique column does not already exist before creating one

```
CREATE FUNCTION [tsc].[GetFirstUniqueColumn] (
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
        ,@columnname as varchar(128) = null
)
RETURNS varchar(128)
AS BEGIN
        declare @FirstUniqueColumn as varchar(128)

        ;with ColumnSelect as (
                SELECT TOP 1 [COLNAME], row_number() OVER ( ORDER BY PKC desc, column_id ) rn
                FROM
                (
                        SELECT
                                 CCU.COLUMN_NAME
                                [COLNAME]
                                ,CASE WHEN LEFT(TC.CONSTRAINT_TYPE,1) = 'P' THEN 1 ELSE 0 END     PKC
                                ,C.column_id

                        FROM            INFORMATION_SCHEMA.TABLE_CONSTRAINTS             TC
                        INNER JOIN      INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE CCU   on
TC.CONSTRAINT_NAME = CCU.CONSTRAINT_NAME
                        INNER JOIN      sys.columns
                C       on C.name = CCU.column_name

                        WHERE TC.CONSTRAINT_SCHEMA      = @schemaname
                        and CCU.CONSTRAINT_SCHEMA       = @schemaname
                        and TC.TABLE_NAME                       = @tablename
                        and TC.CONSTRAINT_TYPE in ('UNIQUE','PRIMARY KEY')
                        and TC.CONSTRAINT_CATALOG = CCU.CONSTRAINT_CATALOG
                        and C.object_id = OBJECT_ID(TC.CONSTRAINT_SCHEMA+'.'+@tablename)
                        and c.is_nullable = 0

                UNION ALL

                        SELECT
                                 ic.name
                                        [COLNAME]
                                ,0
                                                        PKC
                                ,column_id

                        FROM SYS.IDENTITY_COLUMNS ic
                        INNER JOIN sys.tables d on d.object_id = ic.object_id
                        INNER JOIN sys.schemas s on s.schema_id = d.schema_id

                        WHERE s.name = @schemaname
                        and ic.is_nullable = 0
                        and d.name = @tablename
                ) cols
        )
        SELECT @FirstUniqueColumn = COLNAME FROM ColumnSelect WHERE rn = 1

        RETURN(@FirstUniqueColumn)
END
```

## Stored procedures

### CreateUniqueColumn

This procedure is necessary for the audit function to work. If we are unable to distinguish between 2 rows and either one of them change multiple times, how will we track which state applies to which row?

The way it works is to first check if the table already has a unique column using the GetFirstUniqueColumn function. If there isn't already a unique column the procedure checks if the table contains a column called "AutoId" if it does, it renames this column to "AutoId_old" and creates a an identity column called "AutoId" in it's stead.

This procedure should be changed to not rename existing table columns, but instead to change the name of the unique column it's adding to the table, whether that's as a random text string or an incrementing number.

Not sure what the relationship be with the GetFirstUniqueColumn function either, should CreateUniqueColumn be a part of the GetFirstUniqueColumn? As in "If not exist then create" or should they be separate things?

```
CREATE PROCEDURE [tsc].[CreateUniqueColumn]
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
        ,@columnname as varchar(128) = null
        ,@FirstUniqueColumn as varchar(128) OUTPUT
AS
BEGIN
        declare @source as varchar(261)= '[' + @schemaname + '].[' + @tablename + ']'
        declare @ExistingColumn as varchar(128) = (SELECT
tsc.GetFirstUniqueColumn(@schemaname,@tablename,null))
        IF @ExistingColumn is null
        BEGIN
                ;declare @tmpsql as varchar(max)
                ;declare @AutoIdColumn as varchar(50) = isnull(@columnname,'AutoId')

                ;IF (SELECT 1 FROM sys.columns WHERE Name = @AutoIdColumn AND Object_ID = Object_ID(@source))
= 1
                BEGIN
                        ;declare @OriginalAutoIdColumn as varchar(150) = @source+'.'+@AutoIdColumn
                        ;declare @RenamedAutoIdColumn as varchar(128) = @AutoIdColumn+'_old'
                        ;EXEC sp_RENAME @OriginalAutoIdColumn, @RenamedAutoIdColumn, 'COLUMN'
                END
                ;set @tmpsql = ';ALTER TABLE ' + @source + ' ADD ' + @AutoIdColumn + ' int IDENTITY(1,1)
UNIQUE;'
                ;EXEC(@tmpsql)
                ;SELECT @FirstUniqueColumn = @AutoIdColumn
        END
        ELSE BEGIN
                print 'An Identity column [' + @ExistingColumn + '] already exists on ' + @source
                ;SELECT @FirstUniqueColumn = @ExistingColumn
        END
        RETURN 1;
END
```

## CreatePankoTables

This procedure creates tables in the *state* and *delta* schemas so that triggers can be attached

```
CREATE PROCEDURE [tsc].[CreatePankoTables]
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
AS
BEGIN
        /* v17 07-Oct-2019 -- First Refactoring */
        declare @source as varchar(261) = '[' + @schemaname + '].[' + @tablename + ']'
        declare @state as varchar(150) = '[state].[' + @tablename + ']'
        declare @change as varchar(150) = '[delta].[' + @tablename + ']'
        declare @note as varchar(150) = '[note].[' + @tablename + ']'

        declare @tmpsql as nvarchar(max)

        /*****************************
         Has Unique Column?
                NO => Create AutoId
                YES => Store @FirstUniqueColumn
        *****************************/
        declare @FirstUniqueColumn as varchar(128)
        declare @HasUniqueColumn int
        EXEC @HasUniqueColumn = tsc.CreateUniqueColumn @schemaname,@tablename,null, @FirstUniqueColumn OUTPUT

        /*****************************
                CREATE @state
        *****************************/
        --Check if the table has been dropped before and shift it back from the hold.
        ----POTENTIAL BUG!      --The hold will fail where tables are dropped and recreated with different
columns!
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''hold_state.'+ @tablename + ''',''U'') is not null ALTER
SCHEMA state TRANSFER hold_state.[' + @tablename + ']'
        print isnull(@tmpsql,'NULLED!')
        EXEC(@tmpsql)

        ;IF OBJECT_ID(@state,'U') is null
        BEGIN
                set @tmpsql = ''
                --Use same columns and types as @source but without constraints
                set @tmpsql = @tmpsql + ';CREATE TABLE ' + @state + '(' + tsc.GetColumnCreate( @schemaname
,@tablename )
                --Add primary key StateId to @state --Review the clustering of this: I believe the table
should be clustered on the @source PK/UK first
                set @tmpsql = @tmpsql + ', StateId int IDENTITY(1,1) PRIMARY KEY NONCLUSTERED'
                set @tmpsql = @tmpsql + ')'

                print isnull(@tmpsql,'NULLED!')
                EXEC(@tmpsql)
        END

        /*****************************
                CREATE @change
        *****************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''hold_change.'+ @tablename + ''',''U'') is not null ALTER
SCHEMA change TRANSFER hold_change.[' + @tablename + ']'
        print isnull(@tmpsql,'NULLED!')
        EXEC(@tmpsql)

        ;IF OBJECT_ID(@change,'U') is null
        BEGIN
                set @tmpsql = ''
                set @tmpsql = @tmpsql + ';CREATE TABLE ' + @change + ' ( '
                set @tmpsql = @tmpsql + ' ChangeId               int
        IDENTITY(1,1) PRIMARY KEY CLUSTERED'
                set @tmpsql = @tmpsql + ',State                  int                                 NOT NULL'
                set @tmpsql = @tmpsql + ',Step                   int                                 NOT NULL'
                set @tmpsql = @tmpsql + ',ActiveFrom     datetime        NOT NULL'
                set @tmpsql = @tmpsql + ',ActiveTo               datetime        NULL'
                set @tmpsql = @tmpsql + ',Author         varchar(128)    NOT NULL'
                set @tmpsql = @tmpsql + ', CONSTRAINT FK_' + LEFT(@tablename,128-15) + '_Panko FOREIGN KEY
(State) REFERENCES ' + @state + ' (StateId)'
                set @tmpsql = @tmpsql + ')'

                print isnull(@tmpsql,'NULLED!')
                EXEC(@tmpsql)
        END
        /******************************
                CREATE @note
        ******************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''hold_note.'+ @tablename + ''',''U'') is not null ALTER SCHEMA
change TRANSFER hold_note.[' + @tablename + ']'
        print isnull(@tmpsql,'NULLED!')
        EXEC(@tmpsql)


        ;IF OBJECT_ID(@note,'U') is null
        BEGIN
                set @tmpsql = ''
                set @tmpsql = @tmpsql + ';CREATE TABLE ' + @note + ' ( '
                set @tmpsql = @tmpsql + ' NoteId         int                              IDENTITY(1,1)
PRIMARY KEY CLUSTERED'
                set @tmpsql = @tmpsql + ',Change         int                              NOT NULL'
                set @tmpsql = @tmpsql + ',Author         varchar(128)     NOT NULL'
                set @tmpsql = @tmpsql + ',Content                nvarchar(max)    NOT NULL'
                set @tmpsql = @tmpsql + ', CONSTRAINT FK_' + LEFT(@tablename,128-14) + '_ChangeNote FOREIGN
KEY (Change) REFERENCES ' + @change + ' (ChangeId)'
                set @tmpsql = @tmpsql + ')'

                print isnull(@tmpsql,'NULLED!')
                EXEC(@tmpsql)
        END

        /******************************
                ATTACH TRIGGERS
        ******************************/
        --EXEC tsc.CreatePankoTriggers @schemaname, @tablename
END
```

## CreatePankoTriggers

This procedure is run after CreatePankoTables to

```
CREATE PROCEDURE [tsc].[CreatePankoTriggers]
         @schemaname as varchar(128)
        ,@tablename as varchar(128)
AS
BEGIN

        declare @source as varchar(261) = '[' + @schemaname + '].[' + @tablename + ']'
        declare @state as varchar(150) = '[state].[' + @tablename + ']'
        declare @change as varchar(150) = '[delta].[' + @tablename + ']'
        --declare @note as varchar(150) = '[note].[' + @tablename + ']'

        ;IF OBJECT_ID(@source,'U') is null
        BEGIN
                print @source + ' does not exist'
                RETURN 0;
        END
        declare @UniqueColumn                    as varchar(128) = ( SELECT
tsc.GetFirstUniqueColumn(@schemaname, @tablename, null) )
        -- 14-Apr-2020 this variable had to be added for where the UniqueColumn is not the first column in the
table for state deletion inserts
        declare @ColumnListUniqueFirst   as varchar(max) = ( SELECT tsc.GetColumnSelect(@schemaname,
@tablename, null) )
        declare @tmpsql                                  as nvarchar(max)


        IF ( CHARINDEX(@UniqueColumn,@ColumnListUniqueFirst) > 2 )
        BEGIN
                set @ColumnListUniqueFirst = '['+@UniqueColumn+'],'+replace(( SELECT
tsc.GetColumnSelect(@schemaname, @tablename, null) ),',['+@UniqueColumn+']','')
        END
        /*****************************
                ATTACH TRIGGER INSERT to @change -- This updates previous ActiveTo results (The state 
regression issue occurs before this On insert )
        *****************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';CREATE TRIGGER tscUpdateChange_' + @tablename + ' ON ' + @change + ' FOR 
INSERT AS '
        set @tmpsql = @tmpsql + ';BEGIN TRANSACTION '
        set @tmpsql = @tmpsql + ';UPDATE ' + @change + ' SET ActiveTo = NewActiveTo FROM '
        set @tmpsql = @tmpsql + '( SELECT c.ChangeId CID, LEAD(c.ActiveFrom, 1, null) OVER ( PARTITION BY s.' 
+ @UniqueColumn + ' ORDER BY c.ActiveFrom ) NewActiveTo '
        set @tmpsql = @tmpsql + ' FROM '         + @state +  ' s with(nolock) '
        set @tmpsql = @tmpsql + ' INNER JOIN '   + @change + ' c with(nolock) on c.State = s.StateId '
        set @tmpsql = @tmpsql + ') upd WHERE ActiveTo is null and NewActiveTo is not null and CID = ChangeId '
        set @tmpsql = @tmpsql + ';COMMIT TRANSACTION'
        print isnull(@tmpsql,'NULLED!')
        --set @tmpsql = @tmpsql + ';print isnull('''+ replace(@tmpsql,'''','''''') +''',''NULLED!'')'
        EXEC(@tmpsql)

        /*****************************
                ATTACH TRIGGER INSERT to @state -- Is aggregate the only way to increase the step? The most 
efficient?
        *****************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';CREATE TRIGGER tscInsertChange_' + @tablename + ' ON ' + @state + ' FOR 
INSERT AS '
        set @tmpsql = @tmpsql + ';BEGIN TRANSACTION '
        set @tmpsql = @tmpsql + ';INSERT INTO ' + @change + ' (State,Step,ActiveFrom,ActiveTo,Author) '
        set @tmpsql = @tmpsql + ' SELECT i.StateId, 1+isnull(max(c.Step),0), GETDATE(), null, Suser_name()'
        set @tmpsql = @tmpsql + ' FROM inserted i '
        set @tmpsql = @tmpsql + ' INNER JOIN ' + @state + '  s with(nolock) on i.' + @UniqueColumn + ' = s.' + 
@UniqueColumn
        set @tmpsql = @tmpsql + ' LEFT JOIN ' + @change + ' c with(nolock) on c.State = s.StateId'
        set @tmpsql = @tmpsql + ' GROUP BY i.StateId, s.[' + @UniqueColumn + ']'
        set @tmpsql = @tmpsql + ';COMMIT TRANSACTION'
        print isnull(@tmpsql,'NULLED!')
        --set @tmpsql = @tmpsql + ';print isnull('''+ replace(@tmpsql,'''','''''') +''',''NULLED!'')'
        EXEC(@tmpsql)

        /*****************************
                ATTACH TRIGGER INSERT to @source
        *****************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';CREATE TRIGGER tscInsertState_' + @tablename + ' ON ' + @source + ' FOR 
INSERT AS '
        set @tmpsql = @tmpsql + ';BEGIN TRANSACTION '
        set @tmpsql = @tmpsql + ';INSERT INTO ' + @state + ' SELECT * FROM inserted ORDER BY [' + 
@UniqueColumn + ']'
        set @tmpsql = @tmpsql + ';COMMIT TRANSACTION'
        print isnull(@tmpsql,'NULLED!')
        --set @tmpsql = @tmpsql + ';print isnull('''+ replace(@tmpsql,'''','''''') +''',''NULLED!'')'
        EXEC(@tmpsql)

        /*****************************
                ATTACH TRIGGER DELETE to @source
        *****************************/

        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';CREATE TRIGGER tscDeleteState_' + @tablename + ' ON ' + @source + ' FOR 
DELETE AS '
        set @tmpsql = @tmpsql + ';BEGIN TRANSACTION '
        set @tmpsql = @tmpsql + ';INSERT INTO ' + @state + ' (' + @ColumnListUniqueFirst + ') SELECT ' + 
@UniqueColumn + REPLICATE(',null',tsc.NumberOfColumns('input',@tablename)-1) + ' FROM deleted'
        set @tmpsql = @tmpsql + ';COMMIT TRANSACTION'
        print isnull(@tmpsql,'NULLED!')
        --set @tmpsql = @tmpsql + ';print isnull('''+ replace(@tmpsql,'''','''''') +''',''NULLED!'')'
        EXEC(@tmpsql)
        /*****************************
                ATTACH TRIGGER UPDATE to @source
        *****************************/
        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';CREATE TRIGGER tscUpdate_' + @tablename + ' ON ' + @source + ' FOR UPDATE AS
'
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''tempdb..#upd'',''U'') is not null DROP TABLE #upd'
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''tempdb..#existingState'',''U'') is not null DROP TABLE
#existingState'
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''tempdb..#newState'',''U'') is not null DROP TABLE #newState'
        set @tmpsql = @tmpsql + ';BEGIN TRANSACTION ' --#upd Is new values where there was an old value
(inserted inner join deleted).
        --set @tmpsql = @tmpsql + ';SELECT i.* into #upd FROM inserted i INNER JOIN deleted d on d.' +
@UniqueColumn + ' = i.' + @UniqueColumn

        --17-Apr-2020 Only use where values have actually changed
        set @tmpsql = @tmpsql + ';SELECT i.* into #upd FROM inserted i EXCEPT SELECT * FROM deleted d'
        set @tmpsql = @tmpsql + ';SELECT * into #existingState FROM #upd INTERSECT SELECT ' +
tsc.GetColumnSelect('input',@tablename,null) + ' FROM ' + @state
                set @tmpsql = @tmpsql + ';IF (SELECT count(*) FROM #existingState) > 0 '
                set @tmpsql = @tmpsql + ' BEGIN '
                --Removing this insert prevents the [change] from reactivating existing [states]
                set @tmpsql = @tmpsql + ' INSERT INTO ' + @change + ' (State,Step,ActiveFrom,ActiveTo,Author)
'
                set @tmpsql = @tmpsql + ' SELECT B.StateId, 1+isnull(max(c.Step),0), GETDATE(), null,
Suser_name()'
                set @tmpsql = @tmpsql + ' FROM #existingState A '
                set @tmpsql = @tmpsql + ' INNER JOIN ' + @state + '  B with(nolock) on A.' + @UniqueColumn + '
= B.' + @UniqueColumn
                set @tmpsql = @tmpsql + ' LEFT JOIN ' + @change + ' c with(nolock) on c.State = B.StateId'
                        ---16-Apr-20 This equivalent is necessary to find the correct StateId associated with
the value and @UniqueColumn
                set @tmpsql = @tmpsql + ' WHERE ' + tsc.GetABColumnComparison('input',@tablename)
                set @tmpsql = @tmpsql + ' GROUP BY B.StateId, B.[' + @UniqueColumn + ']'
                set @tmpsql = @tmpsql + ' END '
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''tempdb..#existingState'',''U'') is not null DROP TABLE
#existingState'
        set @tmpsql = @tmpsql + ';SELECT * into #newState FROM #upd EXCEPT SELECT ' +
tsc.GetColumnSelect('input',@tablename,null) + ' FROM ' + @state
                set @tmpsql = @tmpsql + ';IF (SELECT count(*) FROM #newState) > 0 '
                set @tmpsql = @tmpsql + ' BEGIN '
                set @tmpsql = @tmpsql + ' INSERT INTO ' + @state + ' SELECT * FROM #newState'
                set @tmpsql = @tmpsql + ' END '
        set @tmpsql = @tmpsql + ';IF OBJECT_ID(''tempdb..#newState'',''U'') is not null DROP TABLE #newState'
        set @tmpsql = @tmpsql + ';DROP TABLE #upd'
        set @tmpsql = @tmpsql + ';COMMIT TRANSACTION'
        print isnull(@tmpsql,'NULLED!')
        --set @tmpsql = @tmpsql + ';print isnull('''+ replace(@tmpsql,'''','''''') +''',''NULLED!'')'
        EXEC(@tmpsql)

        print 'All trigger creation SQL executes before 539 error is thrown on SELECT into'
END
```

## HoldPanko

At one time this procedure (fired by DDL trigger on DROP TABLE)

```
CREATE PROCEDURE [tsc].[HoldPanko]
        @tablename as varchar(128)
AS
BEGIN
/*
        This trigger CAN move dropped tables to a hold schema instead of ~permanently~ deleting the data
        But for the time being it is cascading the drop to state and change tables
*/
        declare @change as varchar(150) = '[delta].' + '[' + @tablename + ']'
        declare @state as varchar(150) = '[state].' + '[' + @tablename + ']'
        declare @note as varchar(150) = '[note].' + '[' + @tablename + ']'
        PRINT 'Processing: ' + @tablename
        declare @tmpsql as varchar(max)

        set @tmpsql = ''
        set @tmpsql = @tmpsql + ';IF OBJECT_ID('''+ @note+ ''',''U'') is not null DROP TABLE ' + @note
        set @tmpsql = @tmpsql + ';IF OBJECT_ID('''+ @change + ''',''U'') is not null DROP TABLE ' + @change
        set @tmpsql = @tmpsql + ';IF OBJECT_ID('''+ @state       + ''',''U'') is not null DROP TABLE ' +
@state

        print @tmpsql
        EXEC(@tmpsql)

        --PRINT 'Moved to hold: ' + @state + ', ' + @change
        PRINT 'Dropped: ' + @state + ', ' + @change
END
```

### RebuildPankoTriggers

This procedure is useful when altering the way that state change table triggers function because it allows you to recreate triggers on tables that have already been created to update them – this has the “input” schema hardcoded which will have to change. Maybe there should be a table in the tsc schema that keeps track of the audited tables.

```
CREATE PROCEDURE [tsc].[RebuildPankoTriggers]
AS BEGIN
        declare @tmpsql as varchar(max)
        ;with TriggerAction as (
                SELECT   'DROP TRIGGER [' + s.name + '].[' + o.name + '];' [Remove]
                                ,'EXEC tsc.CreatePankoTriggers @schemaname = ''input'', @tablename = ''' +
t.name + ''';' [ReAdd]
                FROM sys.objects        o
                INNER JOIN sys.schemas  s on s.schema_id = o.schema_id
                INNER JOIN sys.tables   t on t.object_id = o.parent_object_id
                WHERE o.type = 'TR'
                and o.name like 'tsc%'
        )

        SELECT @tmpsql = (
                SELECT x
                        FROM (
                                                SELECT [Remove] x       FROM TriggerAction GROUP BY [Remove]
                        UNION ALL       SELECT ReAdd            FROM TriggerAction GROUP BY ReAdd
                        ) actions
        FOR XML PATH(''), TYPE).value('.', 'varchar(max)')

        EXEC(@tmpsql)
END
```

## DDL Triggers

These are the database triggers that are currently used to monitor tables being created and dropped.

### ManifestPanko

This trigger detects if a table has been created in the “input” schema, and then executes the CreatePankoTables procedure on it, followed by the CreatePankoTrableTriggers. This has just got me thinking that “Manifest” is a bad name for the frist procedure – it should probably be something like “CreatePankos”

```
CREATE TRIGGER [ManifestPanko] ON DATABASE
AFTER CREATE_TABLE
AS
        declare @schemaname as varchar(128) = 'input'
        ;IF EVENTDATA().value('(/EVENT_INSTANCE/SchemaName)[1]','varchar(128)') <> @schemaname RETURN;
        PRINT '============= ManifestPanko ================'
        declare @tablename as varchar(128) =
EVENTDATA().value('(/EVENT_INSTANCE/ObjectName)[1]','nvarchar(max)')

        EXEC tsc.CreatePankoTables @schemaname, @tablename;
        /*****************************
                ATTACH TRIGGERS
        *****************************/
        EXEC tsc.CreatePankoTriggers @schemaname, @tablename
GO

ENABLE TRIGGER [ManifestPanko] ON DATABASE
GO
```

### HoldPanko

## Products

- Tables
    - *[delta]*. Change table
    - *[state]*. State table
        - *(StateID IDENTITY(1,1) PK NONCLUSTERED)*

- DDL Table Triggers
    - *[delta]*. Change table
        - tscUpdateChange_
    - *[state]*. State table
        - tscInsertChange_
    - Target table
        - tscInsertState_
        - tscDeleteState_
        - tscUpdate_

## How they fit together