Auto-suggesting foreign keys and data model archaeology

**By** [**David Poole**](http://www.sqlservercentral.com/Authors/Articles/David_Poole/3226/)**, 2018/07/06 (first published: 2016/01/26)**

**FROM:** [**http://www.sqlservercentral.com/articles/Foreign+Keys+(FK)/136128/?utm\_source=SSC&utm\_medium=pubemail**](http://www.sqlservercentral.com/articles/Foreign+Keys+(FK)/136128/?utm_source=SSC&utm_medium=pubemail)

Steve Jones posted a request for an article on [finding missing foreign key constraints and fixing them](http://www.sqlservercentral.com/Forums/Topic1650014-2824-1.aspx).  This was serendipitous timing as the capability Steve was requesting would be ideal in meeting my challenges I was about to face in a data warehouse migration.

* Systems were unsupported
* Documentation sparse or non-existent
* Data models are unknown or poorly understood
* Few (if any) people retain comprehensive technical knowledge of that system.

An understanding of the data model for a system reveals much about the system itself and how it is intended to work.  If you can derive the foreign keys then you can use your tool of choice for reverse engineering a schema diagram.  Of course, if a picture paints a thousand words then a schema diagram is pure oratory

This topic sounded like a straight forward problem however, as you will see, producing a solution took a lot of thought and hard work.

**Things to consider before starting**

There are a number of things to consider when beginning this type of project.

Data compliance

Given the number of published data breaches in 2015, your starting point should be to restore a copy of the database in question into an ***appropriately secure*** environment to carry out your investigation.

Always check if there are any regulatory or legal restrictions on where you restore the database in question.  If you are restoring an HR database or a financial services database you might be required to use a specific and restricted environment and not simply restore the database onto a development machine.

How big is the problem?

The first questions I want to answer are as follows:

1. Will the diagramming/modelling tool be overwhelmed by the number of objects in the database?
2. Will my database yield useful information to the diagramming/modelling tool?

These answers can be found by running a few basic queries on the system tables.

**WARNING**:  I have not used INFORMATION\_SCHEMA views due the Microsoft warning shown below:

*Do not use INFORMATION\_SCHEMA views to determine the schema of an object. The only reliable way to find the schema of a object is to query the sys.objects catalog view.*

*Will the diagramming tool be overwhelmed?*

The query shown below will identify how many objects could  be included in the diagram and whether we have any convenient pagination based on schemas.  There are two points to note about the query below:

* The use of the WITH ROLLUP clause to produce an overall total number of objects
* The use of the GROUPING function to indicate the row that is the overall total number of objects.

;WITH cte(SchemaName)

AS

(

SELECT OBJECT\_SCHEMA\_NAME(T.object\_id)

FROM sys.objects AS T

WHERE T.type='U'

)

SELECT

COALESCE(SchemaName, ' TOTAL NUMBER OF TABLES') AS SchemaName,

COUNT(\*) AS TablesInSchema,

GROUPING(SchemaName) AS IsTotal

FROM cte

GROUP BY SchemaName

WITH ROLLUP

ORDER BY SchemaName

When run on the Adventureworks2014 database this will produce a recordset similar to the one below:-

|  |  |  |
| --- | --- | --- |
| **SchemaName** | **TablesInSchema** | **IsTotal** |
| TOTAL NUMBER OF TABLES | 72 | 1 |
| dbo | 4 | 0 |
| HumanResources | 6 | 0 |
| Person | 13 | 0 |
| Production | 25 | 0 |
| Purchasing | 5 | 0 |
| Sales | 19 | 0 |

*Will the database yield useful information?*

You won’t get much value from pointing a diagramming tool at a database that does not contain foreign key relationships or for which the relationships are sparse.

The query below consists of two common table expressions and a query that utilises them.

* FKTables produces a distinct list of tables that either reference or are referenced by a foreign key relationships
* UserTables is simply a list of user tables

;WITH FKTables (FQTableName) AS (

SELECT OBJECT\_SCHEMA\_NAME(fkeyid)+'.'+OBJECT\_NAME(fkeyid)

FROM sys.sysreferences

UNION

SELECT

OBJECT\_SCHEMA\_NAME(rkeyid)+'.'+OBJECT\_NAME(rkeyid)

FROM sys.sysreferences

),

UserTables (FQTableName) AS (

SELECT OBJECT\_SCHEMA\_NAME(object\_id)+'.'+OBJECT\_NAME(object\_id)

FROM sys.objects

WHERE type='U'

)

SELECT COUNT(U.FQTableName) AS TotalUserTables,

COUNT(F.FQTableName) AS TablesInARelationship

FROM UserTables AS U

LEFT JOIN FKTables AS F

ON U.FQTableName=F.FQTableName

Whereas COUNT(\*) would count all records, COUNT(<fieldname>) only counts the non-null entries.  Tables that do not reference or are not referenced by a foreign key will not be in the query results for the FKTables CTE so the LEFT JOIN will give a NULL field state that will not be counted.

If we run the query against the **Adventureworks2014** database then we get the following results.

|  |  |
| --- | --- |
| **TotalUserTables** | **TablesInARelationship** |
| 71 | 67 |

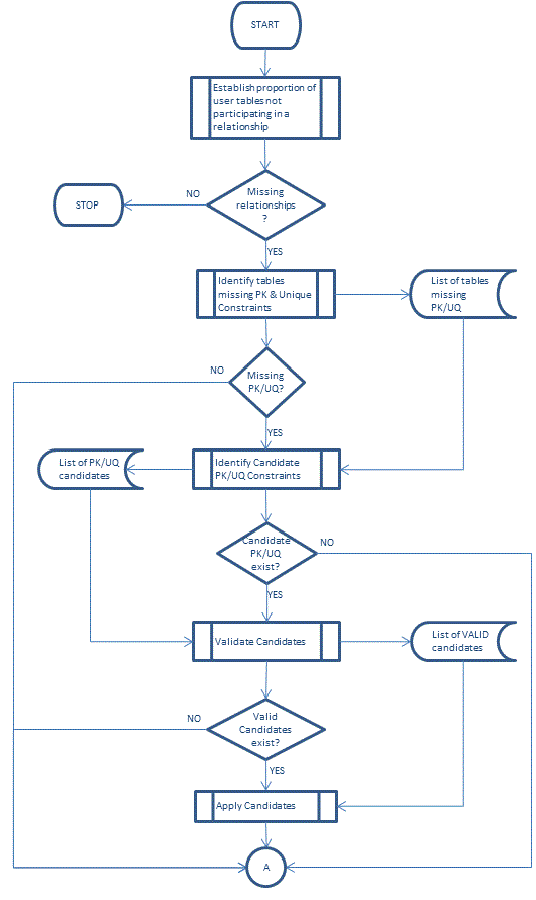
If the figure for TablesInARelationship is a small proportion of the figure for TotalUserTables then pointing a diagramming tool at the database will produce little of value.

Starting off with a high level plan

The next is to take a step back and think through a plan of action. I thought that this could be broken down into a few simple challenges.

* Identify missing primary keys and unique constraints
* Identify the number of fields in the key
* Identify naming conventions for primary key fields and the equivalent in other tables
* Identify candidate FK where a table contains the equivalent fields as those participating in a PK
* Discount those where there is already an FK relationship
* Test the data to precheck violations
* Generate PK & FK scripts through automated means

The flow chart below illustrates my initial plan for identifying primary keys.



We need to establish whether we have the base requirements for adding candidate foreign key constraints (Primary or Unique constraints)?

As this is a copy of a production database we are free to add any foreign key constraints and enabling objects.

Identifying tables without primary or unique constraints

We know that for a foreign key relationship to be created the field(s) within the referenced table must participate in some form of unique constraint, usually a primary key.

**WARNING:**  If the table has a unique index but not a unique constraint then a foreign key relationship cannot be created.

The query to find tables without the required constraints is shown below.

;WITH PK AS (

SELECT parent\_object\_id

FROM sys.objects

WHERE type IN('PK','UQ')

)

SELECT

OBJECT\_SCHEMA\_NAME(T.object\_id)

+ '.'

+ OBJECT\_NAME(T.object\_id) AS FQ\_TableName

FROM sys.objects AS T

LEFT JOIN PK

ON PK.parent\_object\_id = T.object\_id

WHERE

T.type='U'

AND PK.parent\_object\_id IS NULL

If you are lucky there should be very few tables in the results of this query.

But what do you do if tables are listed?

Identifying candidate unique Keys

Devising an automated test can be more error prone than it would first appear from the properties of the two types of unique constraint shown below:

|  |  |  |
| --- | --- | --- |
| **Type of Constraint** | **Nullable** | **Description** |
| Primary Key | NO | Clustered unless explicitly defined otherwise |
| Unique constraint | YES | Where nullable can have a single null value. |

The pitfalls as I see it are as follows:

|  |  |
| --- | --- |
| **Pitfall** | **Description** |
| Compound keys | A legitimate and useful physical modelling technique.  Massively increases the scope of the automated tests that may have to be carried out before applying a unique constraint.  A compound key can be up to 16 fields so you have 65,535 possible combinations.  Granted that there is a 900 byte limit on any key and the most fields I have seen in a primary key is  five but in any one table that still increases the range of automated tests to carry out. |
| Naming conventions | Most people I know tend to stick to adopt a naming convention for fields they use as primary keys.  A quick look at Adventureworks2014 reveals that this isn’t bomb proof.  HumanResources.Employee has a primary key of BusinessEntityID! |
| Co-incidental uniqueness | Where there is a small amount of data then testing an assumption about uniqueness may give a false positive.  ·         We identify a candidate unique key  ·         Data testing indicates that a unique constraint can be added  ·         Although the data is physically compliant with our assumptions those assumptions are incorrect. |

If we limit ourselves to looking for single field candidates for unique keys then can prioritise the things we will evaluate first.

1. Fields that use identity values
2. Fields that have a default for a sequence
3. A field whose name is ID
4. A field whose name is Code
5. A field whose name is basically the table name suffixed by ID.
6. A field whose name is basically the table name suffixed by Code.
7. The first field in a table.
8. Fields that have a GUID type

Housekeeping objects

Clearly evaluating candidate unique constraints is going to be quite an involved process.  We are going to have to build up the information we need in a series of steps and store the results in tables.

As we do not want to pollute our existing schemas with the objects we need our first step is to build a couple of schemas explicitly to hold our objects.

| **Schema** | **Lifecycle** | **Purpose** |
| --- | --- | --- |
| CandidateKey | Permanent | Holds our toolkit.  This consists of tables, views and procs used to identify and evaluate potential keys. |
| Validate | Transient | Holds tables used to validate candidates for unique keys. |

The code below creates the schemas we need.

IF NOT EXISTS(SELECT \* FROM sys.schemas WHERE name='CandidateKey')

BEGIN

EXEC('CREATE SCHEMA CandidateKey')

PRINT 'SCHEMA CREATED: CandidateKey'

END

ELSE

PRINT 'SCHEMA EXISTS: CandidateKey'

GO

IF NOT EXISTS(SELECT \* FROM sys.schemas WHERE name='Validate')

BEGIN

EXEC('CREATE SCHEMA Validate')

PRINT 'SCHEMA CREATED: Validate'

END

ELSE

PRINT 'SCHEMA EXISTS: Validate'

GO

As I intend the Validate schema to hold transient objects I will also need a stored procedure that can clean out any table specific evaluation tables that might exist in that schema.

Under normal circumstances I would parameterise my code to maximize reuse however in the case of a destructive procedure I deliberately hard code it to restrict it to my Validate schema.

IF EXISTS(SELECT \* FROM sys.objects WHERE type='P' AND name='FlushValidateSchema')

BEGIN

DROP PROC CandidateKey.FlushValidateSchema

PRINT 'PROC DROPPED: CandidateKey.FlushValidateSchema'

END

GO

CREATE PROC CandidateKey.FlushValidateSchema

AS

SET NOCOUNT ON

DECLARE @ValidateSchemaObjects TABLE (TableID INT NOT NULL)

INSERT INTO @ValidateSchemaObjects (TableID)

SELECT object\_id

FROM sys.objects AS O

WHERE O.schema\_id=SCHEMA\_ID('Validate')

AND O.name LIKE 'T[0-9][0-9][0-9][0-9]%'

ORDER BY object\_id

DECLARE @TableID INT

DECLARE @SQL VARCHAR(MAX)

SET @TableID = 0

WHILE @TableID IS NOT NULL

BEGIN

SELECT @TableID=MIN(TableID)

FROM @ValidateSchemaObjects

WHERE TableID>@TableID

IF @TableID IS NOT NULL

BEGIN

SET @SQL = 'DROP TABLE '

+ QUOTENAME(OBJECT\_SCHEMA\_NAME(@TableID))

+ '.'

+ QUOTENAME(OBJECT\_NAME(@TableID))

PRINT @SQL

EXEC (@SQL)

END

END

GO

IF @@ERROR=0

PRINT 'PROC CREATED: CandidateKey.FlushValidateSchema'

GO

I want to store the results of the prioritisation so I will create a table in a CandidateKey schema as follows:-

IF EXISTS(SELECT \* FROM sys.objects WHERE object\_id=OBJECT\_ID('CandidateKey.CandidateUniqueConstraints'))

BEGIN

DROP TABLE CandidateKey.CandidateUniqueConstraints

PRINT 'TABLE DROPPED: CandidateKey.CandidateUniqueConstraints'

END

GO

CREATE TABLE CandidateKey.CandidateUniqueConstraints (

TableID INT NOT NULL,

KeyPriority TINYINT NOT NULL,

IsValidated BIT NOT NULL

CONSTRAINT DF\_CandidateUniqueConstraints\_IsValidated DEFAULT(0),

Is\_Nullable BIT NOT NULL

ValidationTable AS 'T'+CAST(TableId AS SYSNAME),

ColumnName SYSNAME NOT NULL,

CONSTRAINT PK\_CandidateUniqueConstraints

PRIMARY KEY CLUSTERED (TableID, ColumnName)

)

GO

IF @@ERROR=0

PRINT 'TABLE CREATED: CandidateKey.CandidateUniqueConstraints'

GO

Priority One - Identifying Identity columns

This is a very simple query to execute. This finds all the tables and columns with the identity property.

INSERT INTO CandidateKey.CandidateUniqueConstraints(TableID, ColumnName, KeyPriority,Is\_Nullable)

SELECT

TableId=object\_id,

ColumnName=name,

KeyPriority=1,

Is\_Nullable

FROM sys.columns

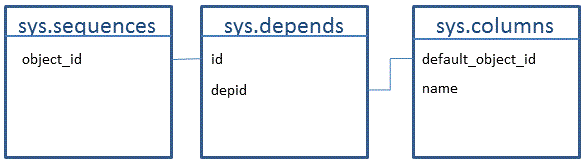
WHERE is\_identity=1

AND OBJECTPROPERTY(object\_id,'IsUserTable')=1

RAISERROR('Candidate Keys with an identity property %d',10,1,@@ROWCOUNT) WITH NOWAIT

**Priority Two - Identifying columns with a sequence default**

This is a bit more complicated and relies on the sys.depends table. Here is the list of tables I will use.



INSERT INTO CandidateKey.CandidateUniqueConstraints(TableID, ColumnName, KeyPriority, Is\_Nullable)

SELECT

TableId=c.object\_id,

ColumnName=c.name,

KeyPriority=2 ,

C.Is\_Nullable

FROM sys.sequences AS SQ

INNER JOIN sysdepends DP

ON SQ.object\_id=DP.depid

INNER JOIN sys.columns AS C

ON C.default\_object\_id = DP.id

WHERE OBJECTPROPERTY(C.object\_id,'IsUserTable')=1

RAISERROR('Candidate Keys using a SEQUENCE %d',10,1,@@ROWCOUNT) WITH NOWAIT

GO

Priorities Three To Eight

It turns out that one nested Common Table Expression query can generate the information we need.

We could have included the priority one identity criteria here however it has been excluded to avoid a situation where priority two sequence defaults lose out to lower priority items.  By running 3 separate queries this situation is avoided.

For simplicity sake we will capture this query in a view.

IF EXISTS(SELECT \* FROM INFORMATION\_SCHEMA.VIEWS WHERE TABLE\_SCHEMA='CandidateKey' AND TABLE\_NAME='CandidatePrimaryKeys')

BEGIN

DROP VIEW CandidateKey.CandidatePrimaryKeys

PRINT 'VIEW DROPPED: CandidateKey.CandidatePrimaryKeys'

END

GO

CREATE VIEW CandidateKey.CandidatePrimaryKeys

AS

WITH CleanTables AS (

SELECT

TableId=O.object\_id,

TableName=O.name,

CleanTableName=CASE WHEN LEFT(O.name,3)='tbl' THEN REPLACE(SUBSTRING(O.name,4,LEN(O.name)),'\_','')

ELSE O.name

END

FROM sys.objects AS O

WHERE O.type='U'

),

PriorityFields AS (

SELECT

O.TableId,

ColumnName=C.name,

Priority=CASE

WHEN C.name='ID' THEN 3

WHEN C.name='Code' THEN 4

WHEN CleanTableName+'ID'=REPLACE(C.name,'\_','') THEN 5

WHEN CleanTableName+'Code'=REPLACE(C.name,'\_','') THEN 6

WHEN C.column\_id=1 THEN 7

WHEN C.system\_type\_id=36 THEN 8

END,

C.Is\_Nullable

FROM sys.columns AS C

INNER JOIN CleanTables AS O

ON C.object\_id = O.TableId

WHERE C.column\_id=1 or C.name like '%id' OR C.name like '%code' or C.system\_type\_id=36

)

SELECT TableID,

ColumnName,

Priority,

Is\_Nullable

FROM PriorityFields

WHERE Priority IS NOT NULL

GO

IF @@ERROR = 0

PRINT 'VIEW CREATED: CandidateKey.CandidatePrimaryKeys'

GO

The CleanTables CTE addresses the practise of prefixing tables with tbl or tbl\_

The PriorityFields CTE assigns priorities as described in our original priority list.  The point to note is the naming convention logic will deal with any of the following combinations of table name and field name.

|  |  |
| --- | --- |
| **Table name** | **Field Name** |
| Tbl\_MyTable  TblMyTable  MyTable | MyTableId  MyTable\_Id  MyTableCode  MyTable\_Code |

By wrapping up our query in a view the query to append data to our CandidateKey.CandidateUniqueConstraints table is straight forward as shown below.

INSERT INTO CandidateKey.CandidateUniqueConstraints(TableID, ColumnName, KeyPriority, Is\_Nullable)

SELECT

SRC.TableId,

SRC.ColumnName,

SRC.KeyPriority,

SRC.Is\_Nullable

FROM CandidateKey.CandidatePrimaryKeys AS SRC

LEFT JOIN CandidateKey.CandidateUniqueConstraints AS TARG

ON SRC.TableID = TARG.TableID

AND SRC.ColumnName = TARG.ColumnName

WHERE TARG.TableID IS NULL

RAISERROR('Candidate Keys with column properties %d',10,1,@@ROWCOUNT) WITH NOWAIT

GO

I would wrap all 3 queries up into a single stored proc called CandidateKey.GetCandidateKeys

Validating uniqueness

We now need to validate unique values in columns, which takes a bit of work.

How to identify uniqueness

If COUNT(\*)and COUNT(DISTINCT <fieldname>) are equal and greater than zero will tell us if the field name in question contains unique values.

What is needed is a stored proc that allows me to pass in the object\_id for a table identified in the CandidateKey.CandidateUniqueConstraints table.

IF EXISTS(SELECT \* FROM sys.objects WHERE type='P' AND name='AssessCandidateUniqueness')

BEGIN

DROP PROC CandidateKey.AssessCandidateUniqueness

PRINT 'PROC DROPPED: CandidateKey.AssessCandidateUniqueness'

END

GO

CREATE PROC CandidateKey.AssessCandidateUniqueness

@TableID INT

AS

SET NOCOUNT ON

DECLARE @SQL VARCHAR(MAX)

SELECT @SQL=COALESCE(@SQL+',','')

+ 'CAST(CASE WHEN COUNT(\*)>0 AND COUNT(\*)-COUNT(DISTINCT '

+ ColumnName

+ ') =0 THEN 1 ELSE 0 END AS BIT) AS '

+ ColumnName

FROM CandidateKey.CandidateUniqueConstraints

WHERE TableID=@TableID

ORDER BY KeyPriority

SET @SQL='SELECT COUNT(\*) AS TotalRows,' + @SQL

+ ' INTO Validate.T'

+ CAST(@TableID AS SYSNAME)

+ ' FROM '

+ OBJECT\_SCHEMA\_NAME(@TableID)+'.'+OBJECT\_NAME(@TableID)

PRINT @SQL

EXEC(@SQL)

GO

IF @@ERROR=0

PRINT 'PROC CREATED: CandidateKey.AssessCandidateUniqueness'

GO

Safety Considerations when identifying uniqueness

**WARNING:** Running COUNT(DISTINCT <fieldname>) across a number of fields on a large table can be extremely expensive.  For this reason it is wise to run a simple query to determine the number of records in our database tables.

SELECT OBJECT\_SCHEMA\_NAME(object\_id) AS SchemaName,

OBJECT\_NAME(object\_id) AS TableName,

SUM(rows) AS RowCountEstimate

FROM sys.partitions

WHERE index\_id in(0,1)

AND OBJECTPROPERTY(object\_id,'IsUserTable')=1

GROUP BY object\_id

ORDER BY 3 DESC

Evaluating a database safely

I need a stored proc that will loop through the candidate tables executing the CandidateKey.AddessCandidateUniqueness procedure.  This stored procedure should have safety logic to check that any qualifying tables will have their row count checked against the specified thresholds as follows:

* More rows than the ceiling row count will raise an error.
* Fewer rows than the floor row count will be ignored.

IF EXISTS(SELECT \* FROM sys.objects WHERE type='P' AND name='ExecuteCandidateUniqueness')

BEGIN

DROP PROC CandidateKey.ExecuteCandidateUniqueness

PRINT 'PROC DROPPED: CandidateKey.ExecuteCandidateUniqueness'

END

GO

CREATE PROC CandidateKey.ExecuteCandidateUniqueness

@ThresholdFloorRowCount BIGINT=0, --##PARAM @ThresholdFloorRowCount The minimum number of records for which a uniqueness assessment will take place.

@ThresholdCeilingRowCount BIGINT=10000 --##PARAM @@ThresholdCeilingRowCount The maximum number of records for which a uniqueness assessment will take place.

AS

SET NOCOUNT ON

DECLARE

@TableID INT,

@RowCount BIGINT,

@FQTableName SYSNAME

SET @TableID=0

WHILE @TableID IS NOT NULL

BEGIN

SELECT @TableID=MIN(TableID)

FROM CandidateKey.CandidateUniqueConstraints

WHERE TableID>@TableID

IF @TableID IS NOT NULL

BEGIN

SET @FQTableName=QUOTENAME(OBJECT\_SCHEMA\_NAME(@TableID))+'.'+QUOTENAME(OBJECT\_NAME(@TableID))

SELECT @RowCount=SUM(rows)

FROM sys.partitions

WHERE object\_id = @TableID

AND index\_id IN(0,1)

IF @RowCount > @ThresholdCeilingRowCount

BEGIN

RAISERROR('THRESHOLD %I64d EXCEEDED: %s = %I64d ',10,1,@ThresholdCeilingRowCount,@FQTableName,@RowCount) WITH NOWAIT

END

ELSE

BEGIN

IF @RowCount >= @ThresholdFloorRowCount

exec CandidateKey.AssessCandidateUniqueness @TableID

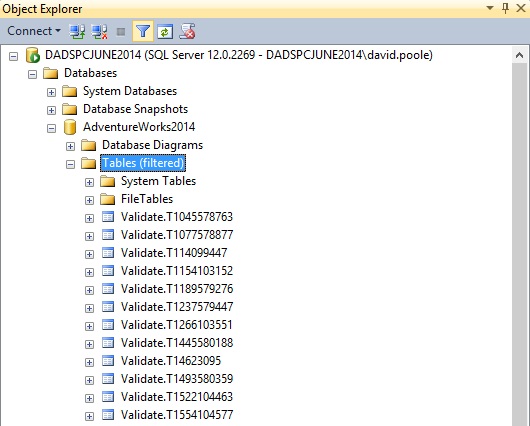
END

END

END

GO

By the time we have executed our CandidateKey.ExecuteCandidateUniqueness and swept up all relevant database objects our Validate schema will look something like the following:



The structure of a Validate table

The table below recaps the naming convention of the tables in the Validate schema.

|  |  |  |
| --- | --- | --- |
| **Original Table** | **Object\_Id** | **Validate table** |
| Sales.SalesOrderDetail | 1154103152 | T1154103152 |

If we look at the table the structure is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **TotalRows** | **SalesOrderDetailID** | **SalesOrderID** | **rowguid** |
| 121317 | 1 | 0 | 1 |

This tells us that SalesOrderID is not a valid candidate for a unique key but the other two fields are based on an evaluation of all the rows in the table.

This structure of the table came about due to what we captured in our CandidateKey.CandidateUniqueConstraints table forSales.SalesOrderDetail.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TableID** | **KeyPriority** | **IsValidated** | **ValidationTable** | **ColumnName** |
| 1154103152 | 1 | 0 | T1154103152 | SalesOrderDetailID |
| 1154103152 | 7 | 0 | T1154103152 | SalesOrderID |
| 1154103152 | 8 | 0 | T1154103152 | rowguid |

Our next step is to update the IsValidated field with the appropriate flag from our Validate. T1154103152 table.

We need to turn our T1154103152 table into a recordset as follows:

|  |  |  |
| --- | --- | --- |
| **TableId** | **ColumnName** | **IsValidated** |
| 1154103152 | SalesOrderDetailID | 1 |
| 1154103152 | SalesOrderID | 0 |
| 1154103152 | rowguid | 1 |

Fortunately we have the T-SQL UNPIVOT clause to help us.

SELECT TableId, ColumnName, IsValidated

FROM

(SELECT 1154103152 AS TableId, SalesOrderDetailID, SalesOrderID, rowguid

FROM validate.T1154103152) p

UNPIVOT

(

IsValidated FOR ColumnName IN (SalesOrderDetailID, SalesOrderID, rowguid)

)AS unpvt;

GO

We need to do this for all the T<object\_id> tables in the Validate schema and record the results.

Table to record the results of the UNPIVOT function.

As our results will be used to update CandidateKey.  We will create a transient shadow table in the Validate schema

IF EXISTS(SELECT \* FROM sys.objects WHERE object\_id=OBJECT\_ID('Validate.CandidateUniqueConstraints'))

BEGIN

DROP TABLE Validate.CandidateUniqueConstraints

PRINT 'TABLE DROPPED: Validate.CandidateUniqueConstraints'

END

GO

CREATE TABLE Validate.CandidateUniqueConstraints (

TableId INT NOT NULL,

ColumnName SYSNAME NOT NULL,

IsValidated BIT NOT NULL,

CONSTRAINT PK\_Validate\_CandidateUniqueConstraints

PRIMARY KEY CLUSTERED (TableId, ColumnName)

)

GO

IF @@ERROR=0

PRINT 'TABLE CREATED: Validate.CandidateUniqueConstraints'

GO

Procs to generate and execute the UNPIVOT functionality.

We need two stored procedures to keep record the evaluation of our candidate keys

* Generate and execute the required UNPIVOT query
* Loop through the Validate.T<Object\_Id> tables executing the above query.

The first query is as follows:

CREATE PROC CandidateKey.ExtractValidatedCandidates

@TableName SYSNAME -- The name of the Validate.T<Object\_id> table

AS

SET NOCOUNT ON

--Must be a table matching the desired pattern in the Validate schema.

IF @TableName NOT LIKE 'T[0-9][0-9][0-9]%' OR NOT EXISTS (SELECT 1 FROM sys.objects WHERE name = @TableName AND schema\_id=SCHEMA\_ID('Validate'))

BEGIN

RAISERROR('Table %s is not a validation table',16,1,@TableName) with nowait

RETURN 1

END

DECLARE

@TableID VARCHAR(10),

@ValidationTableID INT ,

@ColumnList VARCHAR(MAX),

@SQL VARCHAR(MAX)

SET @TableID = RIGHT(@TableName,LEN(@TableName)-1) --The numeric part of the Validate.T<Object\_id> table name

SET @ValidationTableID = OBJECT\_ID('Validate.'+@TableName) -- The object\_id of the actual Validate.T<Object\_id> table name

SELECT @ColumnList=COALESCE(@ColumnList+',','')

+ name

FROM sys.columns AS C

WHERE object\_id=@ValidationTableID

AND name<>'TotalRows'

ORDER BY C.column\_id

SET @SQL='INSERT INTO Validate.CandidateUniqueConstraints(TableId, ColumnName, IsValidated)

SELECT TableId, ColumnName, IsValidated

FROM (

SELECT '

+ @TableID

+ ' AS TableId, '

+ @ColumnList

+ ' FROM Validate.'+@TableName+') p

UNPIVOT (

IsValidated FOR ColumnName IN ('

+ @ColumnList

+ ') )AS unpvt'

EXEC(@SQL)

GO

IF @@ERROR=0

PRINT 'PROC CREATED: CandidateKey.ExtractValidatedCandidates'

GO

The 2nd stored procedure is straight forward.

CREATE PROC CandidateKey.SetCandidateValidity

AS

SET NOCOUNT ON

DECLARE @TableName SYSNAME='',

@SchemaID int

SET @SchemaID=SCHEMA\_ID('Validate')

TRUNCATE TABLE Validate.CandidateUniqueConstraints

WHILE @TableName IS NOT NULL

BEGIN

SELECT @TableName = MIN(name)

FROM sys.objects

WHERE name>@TableName

AND schema\_id=@SchemaID

AND type='U'

AND name LIKE 'T[0-9][0-9][0-9][0-9]%'

IF @TableName IS NOT NULL

BEGIN

EXEC CandidateKey.ExtractValidatedCandidates @TableName

END

END

UPDATE DEST

SET DEST.IsValidated=SRC.IsValidated

FROM Validate.CandidateUniqueConstraints AS SRC

INNER JOIN CandidateKey.CandidateUniqueConstraints AS DEST

ON SRC.TableID = DEST.TableID

AND SRC.ColumnName = DEST.ColumnName

WHERE SRC.IsValidated<>DEST.IsValidated

RAISERROR ('%d Candidate Keys Updated',10,1,@@ROWCOUNT) WITH NOWAIT

-- Discard any candidates that didn't resolve as PK Candidates

DELETE FROM CandidateKey.CandidateUniqueConstraints WHERE IsValidated=0

RAISERROR ('%d Invalid candidate Keys removed',10,1,@@ROWCOUNT) WITH NOWAIT

GO

Putting it all together

We can assemble the execution of our stored procedures into a repeatable script

-- Flush out existing tables

exec CandidateKey.FlushValidateSchema

-- Grab a list of potential candidates

exec CandidateKey.GetCandidateKeys

-- Evaluate the uniqueness of the candidate fields

exec CandidateKey.ExecuteCandidateUniqueness 0,999999

-- Unpivot the results and flag the potential candidates

exec CandidateKey.SetCandidateValidity

Looking at the recommended keys

We can also build a query to list the ALTER TABLE commands necessary to create the missing primary keys in our database.

-- List the SQL to generate the candidate keys

;WITH PotentialKeys(Likelihood,SchemaName,TableName,ColumnName,KeyPriority) AS (

SELECT

ROW\_NUMBER() OVER(PARTITION BY TableID ORDER BY KeyPriority),

SchemaName=OBJECT\_SCHEMA\_NAME(TableID),

TableName=OBJECT\_NAME(TableID),

ColumnName,

KeyPriority

FROM CandidateKey.CandidateUniqueConstraints

)

SELECT

'ALTER TABLE ' + SchemaName + '.' + TableName

+ ' ADD CONSTRAINT PKCandidate\_' + SchemaName + '\_' + TableName

+ ' PRIMARY KEY (' + ColumnName +')'

FROM PotentialKeys

WHERE Likelihood=1

ORDER BY SchemaName,TableName

GO

When I ran the entire process against the Adventureworks2014 database the process did produce credible primary key candidates.  However, as stated earlier no system is perfect and some anomalies crept in.

|  |  |
| --- | --- |
| **Anomaly** | **Reason** |
| Incidents of the rowguid field | The rowguid field has a particular mechanical significance which means that it gives a false positive.  In the Adventureworks2014 schema it aligns to those tables where the true primary key would actually be a compound key. |
| Person.EmailAddress | An example of coincidental uniqueness.  The true primary key is a compound key of BusinessEntityID and EmailAddressID.  In this case the identity property caused the field to be chosen as a candidate and it did turn out to hold unique values. |
| Person.PersonPhone | Again, coincidental uniqueness.  The true primary key is a compound key of BusinessEntityId, PhoneNumber and PhoneNumberTypeId.  In this case BusinessEntityID was chosen as a candidate because it is the first field in a record. |

With the knowledge that rowguid is a false positive I ran a slightly tweaked version for supplemental unique constraints.

;WITH PotentialKeys(Likelihood,SchemaName,TableName,ColumnName,KeyPriority) AS (

SELECT

ROW\_NUMBER() OVER(PARTITION BY TableID ORDER BY KeyPriority),

SchemaName=OBJECT\_SCHEMA\_NAME(TableID),

TableName=OBJECT\_NAME(TableID),

ColumnName,

KeyPriority

FROM CandidateKey.CandidateUniqueConstraints

)

SELECT

'ALTER TABLE ' + SchemaName + '.' + TableName

+ ' ADD CONSTRAINT UQCandidate\_' + SchemaName + '\_' + TableName+'\_'+ColumnName

+ ' UNIQUE (' + ColumnName +')'

FROM PotentialKeys

WHERE Likelihood>1

AND ColumnName<>'rowguid'

ORDER BY SchemaName,TableName

GO

Again this picked up Person.EmailAddress and identified BusinessEntityID as a valid unique candidate.

This false positive reveals something interesting about the Person.EmailAddress table.

* The two participants in the primary key are both unique in their own right
* The 2nd field is an IDENTITY field which is an odd thing to have in a compound primary key

This is precisely the sort of information that is useful to a data modeller and it indicates that the modelling of the Person.EmailAddress table is something warranting deeper analysis.

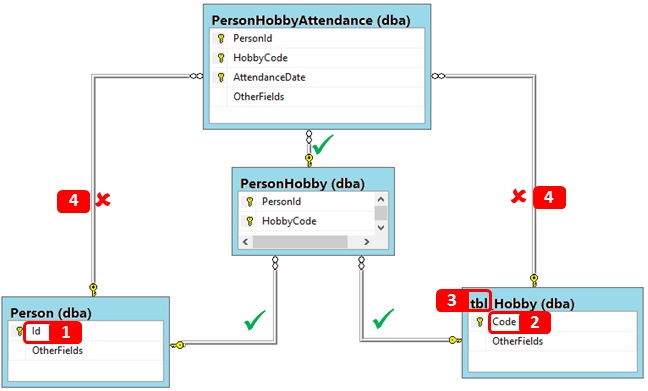
Challenges in Deriving Foreign Key Relationships

As described earlier there are a number of characteristics of fields within tables that defined whether or not they are likely to participate in ***primary keys***.

* Naming conventions
* Identity value
* Use of a sequence
* Data types
* Position within the table structure

Any candidate can be rejected if it does not contain unique values.

With foreign key relationships, short of parsing execution plans, we are much more dependent on naming conventions.  The diagram below illustrates why even a simple schema can present serious challenges.



Let us look at what the challenges are

| **#** | **Challenge** | **Description** |
| --- | --- | --- |
| 1 & 2 | Field whose names have context only within the table | If we have fields such as “Id” and “Code” these only have significance within the table in which they are embedded.  PersonID has its own context, Id does not.  We can add artificial context by prefixing the fieldname with the table name.  When doing so we have to allow for prefixes such as tbl in table names. |
| 3 | Prefixing of table names | If prefixes are consistently applied then this is easy to cater for. Even when not consistently applied if a separator exists between the prefix and the table name this is reasonably straight forward.  If the prefix could be part of a legitimate word then this introduces the need for language parsing which is too advanced a topic for this article. |
| 4 | Undesired relationships | We don’t want to suggest every possible relationship between two entities.  In our example diagram the relationships with the red crosses are superfluous.  They won’t do any harm but will make any schema diagram more crowded.  Remember we are simply trying to establish relationships in a copy system.  There will be no DML activity in the copy system.  To satisfy this we need a mechanism for determining whether a child table already has a parent relationship that includes the candidate attributes. |
| 5 | 1 to 0..1 relationships | NOT SHOWN ON DIAGRAM  Adventureworks2014 has a number of tables with a primary key of BusinessEntityID.  We need a rules based system for determining the correct lineage between such tables. |
| 6 | Pluralisation of object names | NOT SHOWN ON DIAGRAM  If the tbl\_Hobby table was called tbl\_Hobbies then standardising the Code field to HobbyCode would not be possible without introducing language parsing capabilities.  This is sophistication beyond that which we can cover in this article. |

Caveats

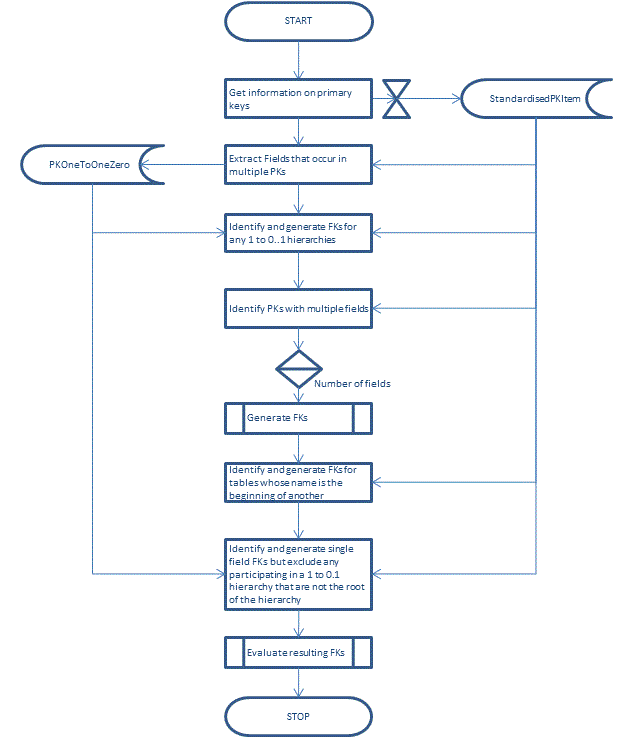
The process of identifying foreign keys is going to depend heavily on naming conventions and establishing rules.

It is difficult enough defining the rules in sufficient detail to attempt to build an automated process let alone building that process.

A naming convention based ruleset will fail if the database is weak in following naming conventions.  Even with a strong naming convention it is easy to get false positives with relationships and also to relate objects that should not be related.

High level process

It helps to have mapped out the approach at a high level to act as a road map.



Prioritising the candidate foreign keys

From our table of challenges we know we need mechanisms for the following:

* Matching on standardised names for table and field objects
* Identifying where a suitable relationship already exists so as not to create another
* A mechanism for determining the master in a 1 to 0..1 relationship
* A mechanism for prioritising the order in which relationships are created.

If we started to create relationships from our tbl\_Hobby table we would get a relationship to both PersonHobby and PersonHobbyAttendance.

If we started to create relationships in reverse order of the number of fields participating in a primary key then we would get the following:

1. A relationship from PersonHobby to PersonHobbyAttendance
2. A relationship from tbl\_Hobby to PersonHobby
3. NO RELATIONSHIP from tbl\_Hobby to PersonHobbyAttendance because the first relationship already contains the HobbyCode fields in a relationship.

Clearly the order in which foreign keys are created is going to be important.

Capturing Primary Key Information

My starting point is to capture information about the primary keys in the database and standardise their field and column names.

CREATE TABLE CandidateKey.StandardisedPKItems (

Object\_id INT NOT NULL ,

ColumnId INT NOT NULL,

DataTypeId INT NOT NULL,

SchemaName SYSNAME NOT NULL,

OriginalObjectName SYSNAME NOT NULL,

StandardisedObjectName SYSNAME NOT NULL,

OriginalColumnName SYSNAME NOT NULL,

StandardisedColumnName SYSNAME NOT NULL,

CONSTRAINT PK\_StandardisedPKItems PRIMARY KEY CLUSTERED (Object\_id,OriginalColumnName)

)

GO

The stored procedure to populate the table is shown below.

CREATE PROC CandidateKey.GetStandardisedPKItems

AS

SET NOCOUNT ON

TRUNCATE TABLE CandidateKey.StandardisedPKItems

INSERT INTO CandidateKey.StandardisedPKItems

SELECT

O.Object\_Id,

C.Column\_Id,

C.system\_type\_id,

OBJECT\_SCHEMA\_NAME(c.object\_id),

O.name,

StandardisedObjectName= REPLACE(CASE WHEN LEFT(O.name,3)='tbl' THEN SUBSTRING(O.name,4,len(O.name)) ELSE O.name END,'\_',''),

OriginalColumnName=C.name,

StandardisedColumnName=REPLACE( -- Deal with primary key fields like Id & code and the tibblers.

CASE WHEN C.name IN('Id','Code') AND LEFT(O.name,3)='tbl' THEN SUBSTRING(O.name,4,len(O.name))+C.name

WHEN C.name IN('Id','Code') AND LEFT(O.name,3)!='tbl' THEN O.name + c.name

ELSE C.name END,

'\_','')

FROM sys.index\_columns AS IC

INNER JOIN sys.indexes AS I

ON IC.object\_id = I.object\_id

AND IC.index\_id = I.index\_id

INNER JOIN sys.columns AS C

ON IC.object\_id = C.object\_id

AND IC.index\_column\_id = C.column\_id

INNER JOIN sys.objects AS O

ON O.object\_id=C.object\_id

WHERE I.is\_primary\_key=1

AND IC.is\_included\_column=0

AND OBJECT\_SCHEMA\_NAME(c.object\_id) NOT IN ('Validate','CandidateKey','dba','sys') -- Exclude mechanical schemas

GO

Dealing with 1 to 0..1 relationships

The first step is to identify tables that share a standardised fieldname.  The common table expression below would identify two fields as occurring across the Adventureworks2014 database

* BusinessEntityID
* TransactionID

;WITH MultiField AS ( -- Single field primary keys that occur more than once i.e. 1 to 0..1 relationships.

SELECT pk.StandardisedColumnName

FROM CandidateKey.StandardisedPKItems AS PK -- Table to contain standardised names of primary key artefacts

INNER JOIN CandidateKey.PKFieldCount AS PKC -- View to determine number of fields in a primary key.

ON pk.Object\_id = PKC.Object\_id

WHERE PKC.PKFieldCount=1

GROUP BY PK.StandardisedColumnName

HAVING COUNT(\*)>1

)

Identifying rules for prioritising tables

We also need to define the rules that will prioritise which table is the master table.

|  |  |  |
| --- | --- | --- |
| **Priority** | **Rule** | **Rationale** |
| 1 | Original field name is ID | This cannot be reduced further. |
| 2 | Original field name is Code | Again this cannot be reduced further |
| 3 | Standardised table name is the beginning of the field name | We are looking at <tablename>id, <tablename>code etc. |
| 4 | Schema name is same as the standardised object name | This assumes that a table name with the same name as the schema in which it resides is of primary importance. |
| 5 | Does not fit any of the preceding rules. | Every field should have a priority value. |

If we could apply these rules to tables that share a primary key then we should have some form of hierarchy for building relationships.  The prioritised list of tables will also have other uses so we will store these in a table defined as follows:

CREATE TABLE CandidateKey.PKOneToOneZero (

Object\_Id INT NOT NULL

CONSTRAINT PK\_PKOneToOneZero PRIMARY KEY CLUSTERED,

TablePriority TINYINT NOT NULL,

SchemaName SYSNAME NOT NULL,

TableName SYSNAME NOT NULL,

StandardisedTableName SYSNAME NOT NULL,

OriginalColumnName SYSNAME NOT NULL,

StandardisedColumnName SYSNAME NOT NULL

)

GO

The number of fields participating in the primary key is also important in the prioritisation and is supported by a view.

CREATE VIEW CandidateKey.PKFieldCount

AS

SELECT Object\_id, PKFieldCount=COUNT(\*)

FROM CandidateKey.StandardisedPKItems

GROUP BY Object\_id

GO

CREATE PROC CandidateKey.GetPKOneToOneZeroPriority

AS

SET NOCOUNT ON

TRUNCATE TABLE CandidateKey.PKOneToOneZero

;WITH MultiField AS ( -- Single field primary keys that occur more than once i.e. 1 to 0..1 relationships.

SELECT pk.StandardisedColumnName

FROM CandidateKey.StandardisedPKItems AS PK -- Table to contain standardised names of primary key artefacts

INNER JOIN CandidateKey.PKFieldCount AS PKC -- View to determine number of fields in a primary key.

ON pk.Object\_id = PKC.Object\_id

WHERE PKC.PKFieldCount=1

GROUP BY PK.StandardisedColumnName

HAVING COUNT(\*)>1

)

INSERT INTO CandidateKey.PKOneToOneZero(

Object\_Id,

TablePriority,

SchemaName,

TableName,

StandardisedTableName,

OriginalColumnName,

StandardisedColumnName

)

SELECT

PK.Object\_id,

TablePriority=CASE

WHEN PK.OriginalColumnName = 'ID' THEN 1

WHEN PK.OriginalColumnName='Code' THEN 2

WHEN PATINDEX(StandardisedObjectName+'%',PK.StandardisedColumnName)=1 THEN 3

WHEN OBJECT\_SCHEMA\_NAME(PK.object\_id) = object\_name(PK.object\_id) THEN 4

ELSE 5

END,

SchemaName=OBJECT\_SCHEMA\_NAME(PK.object\_id),

TableName=object\_name(PK.object\_id),

StandardisedObjectName,

OriginalColumnName,

StandardisedColumnName

FROM CandidateKey.StandardisedPKItems AS PK

INNER JOIN CandidateKey.PKFieldCount AS PKC

ON PK.Object\_id = PKC.Object\_id

WHERE PK.StandardisedColumnName IN (SELECT StandardisedColumnName FROM MultiField)

AND PKC.PKFieldCount=1

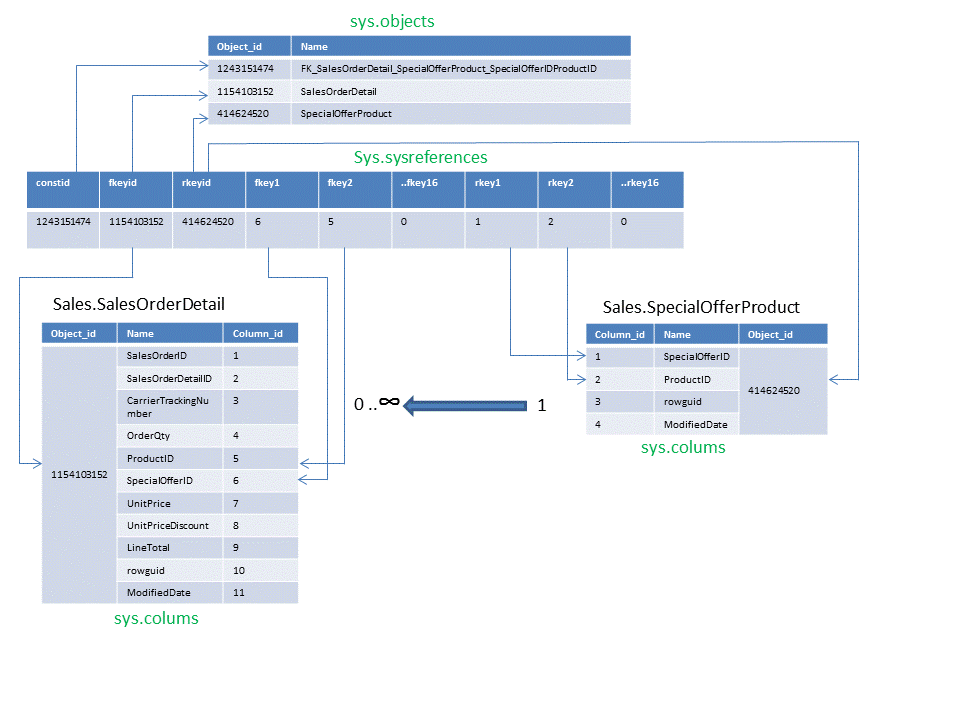
ORDER BY StandardisedColumnName

GO

Function to return fields in an existing relationships

If we are proposing to create a relationship from a parent table to a child so we want to make sure that the child does not already have an existing parent relationship involving any fields that we propose to use in our relationship.

We need our function to retrieve any fields for the given child table that already participate in a relationship.  To do this we need an understanding of how the sys.sysreferences table works.  The diagram uses data within sys.objects, sys.columns and sys.sysreferences to illustrate the key mechanisms.



For a given object\_id our function has to unpivot the sys.sysreferences table and return any fields that are already used in a relationship.

CREATE FUNCTION CandidateKey.GetFieldsUsedInExistingFK(

@ChildObjectId INT

)

RETURNS @ChildFieldList TABLE (

StandardisedColumnName SYSNAME NOT NULL

)

AS

BEGIN

--Unpivot the table.

;WITH CurrentFK (PARENT\_object\_id,CHILD\_object\_id,column\_id)AS (

SELECT DISTINCT rkeyId,fkeyid, column\_id

FROM

(SELECT rkeyid,fkeyid,

fkey1,fkey2,fkey3,fkey4,

fkey5,fkey6,fkey7,fkey8,

fkey9,fkey10,fkey11,fkey12,

fkey13,fkey14,fkey15,fkey16

FROM sysreferences

WHERE fkeyid=@ChildObjectId) p

UNPIVOT

(column\_id FOR KeyID IN

(fkey1,fkey2,fkey3,fkey4,

fkey5,fkey6,fkey7,fkey8,

fkey9,fkey10,fkey11,fkey12,

fkey13,fkey14,fkey15,fkey16)

)AS unpvt

)

INSERT INTO @ChildFieldList(StandardisedColumnName)

SELECT REPLACE(C.name,'\_','')

FROM CurrentFK

INNER JOIN sys.columns AS C

ON CurrentFK.CHILD\_object\_id = C.object\_id

AND CurrentFK.column\_id = C.column\_id

WHERE CurrentFK.column\_id>0 -- Zero means no column at this position

RETURN

END

GO

Generating the 1 to 0..1 keys

We can create a stored procedure to generate the candidate foreign key. There are a four key features to emphasise here

1. To perform a self-join on the prioritised 1 to 0..1 relationships to produce a hierarchy.  As the TablePriority can contain gaps in the priority range we need to create a common table expression that will generate a sequence to align the relationships correctly.
2. To determine the master object in any schema for an attribute and build a relationship from those tables that share a primary key within the same schema but for which we haven’t got a rule
3. We generate the candidate foreign keys using the WITH NOCHECK option.
4. All our foreign keys being with FKCandidate so we can distinguish between the pre-existing relationships and those we generate.

CREATE PROC CandidateKey.GeneratePKOneToOneZeroHierarchy

AS

SET NOCOUNT ON

DECLARE @ProcName VARCHAR(256)=OBJECT\_SCHEMA\_NAME(@@PROCID)+'.'+OBJECT\_NAME(@@PROCID)

PRINT '==============================================='

RAISERROR('PROC: %s',10,1,@ProcName) WITH NOWAIT

DECLARE @SQL VARCHAR(MAX)

DECLARE @CRLF CHAR(2)=CHAR(13)+CHAR(10)

;WITH FKHierarchy AS (

SELECT

FKSequence=ROW\_NUMBER() OVER (PARTITION BY StandardisedColumnName ORDER BY SchemaName,TablePriority ),

Object\_Id, TablePriority, SchemaName, TableName, StandardisedTableName, OriginalColumnName, StandardisedColumnName

FROM CandidateKey.PKOneToOneZero

)

SELECT @SQL = COALESCE(@SQL+';'+@CRLF,'')

+ 'ALTER TABLE '

+ QUOTENAME(c.SchemaName)+'.'+QUOTENAME(C.TableName)

+ ' WITH NOCHECK ADD CONSTRAINT '

+ QUOTENAME('FKCandidate\_'+c.SchemaName+'\_'+C.TableName+'\_'+P.schemaName+'\_'+P.tablename)

+ ' FOREIGN KEY ('

+ C.originalColumnName

+ ') REFERENCES '

+ QUOTENAME(P.SchemaName)+'.'+QUOTENAME(P.TableName)

+ '('

+ p.OriginalColumnName

+ ')'

FROM FKHierarchy AS P

INNER JOIN FKHierarchy AS C

ON p.StandardisedColumnName = c.StandardisedColumnName

AND P.Object\_id<>c.Object\_id

AND P.SchemaName = C.SchemaName

AND P.TablePriority<c.TablePriority

AND P.FKSequence=C.FKSequence-1

AND C.StandardisedColumnName NOT IN(SELECT StandardisedColumnName FROM CandidateKey.GetFieldsUsedInExistingFK( C.Object\_id ))

ORDER BY P.TablePriority

PRINT @SQL

EXEC(@SQL)

GO

Dealing with other relationships

We have dealt with 1 to 0..1 hierarchies within a schema but now we have to consider the one to many relationships and cross schema relationships.

One of the challenges we have to face is that of compound primary keys and also ensuring that we do not get superfluous relationships as described earlier.

**Stored procedure to generate Foreign Keys**

The complexity of what we want to do suggests that a stored procedure that accepts a parent table object\_id and generates the required new relationships for children of that parent.

We can capitalise on the tables we built earlier to hold information on our primary key tables.

CREATE PROC CandidateKey.GetCandidateForeignKey

@TableID INT -- The object\_id of the parent object

AS

SET NOCOUNT ON

DECLARE

@PKFieldCount int,

@SQLReferences VARCHAR(MAX)

DECLARE @FKTable AS TABLE (

Object\_id INT NOT NULL,

ColumnId INT NOT NULL,

StandardisedObjectName SYSNAME NOT NULL ,

OriginalColumnName SYSNAME NOT NULL ,

StandardisedColumnName SYSNAME NOT NULL

)

-- Retrieve the number of fields in the primary key for the table.

SELECT @PKFieldCount = PKFieldCount

FROM CandidateKey.PKFieldCount

WHERE Object\_ID = @TableID

-- Build a comma delimited list of fields in the primary key table.

SELECT @SQLReferences = COALESCE(@SQLReferences+',','')+ OriginalColumnName

FROM CandidateKey.StandardisedPKItems

WHERE Object\_Id=@TableID

ORDER BY ColumnId --Essential for multi-field keys

SET @SQLReferences = ' REFERENCES '+OBJECT\_SCHEMA\_NAME(@TableId)+'.'+OBJECT\_NAME(@TableId)+'('+@SQLReferences+')'

Notice that we order by ColumnId in order as we need to make sure that our FOREIGN KEY fields and REFERENCES fields are going to be in the same order as the order in the primary key.

Recording the foreign key object and fields

We are looking for tables other than the parent that contain all the fields in the parent primary key.  The comparison is done on the standardised field names rather than the actual field names in order to increase the chances of a match.

;WITH FKCandidate

AS (

SELECT

C.object\_id,

C.name,

StandardisedObjectName= REPLACE(CASE WHEN LEFT(O.name,3)='tbl' THEN SUBSTRING(O.name,4,len(O.name)) ELSE O.name END,'\_',''),

StandardisedColumnName=REPLACE(C.name,'\_','')

FROM sys.columns AS C

INNER JOIN sys.objects AS O

ON C.object\_id = O.object\_id

WHERE REPLACE(C.name,'\_','') in (SELECT StandardisedColumnName FROM CandidateKey.StandardisedPKItems WHERE object\_id=@TableID) -- Primary key fields

AND C.object\_id<>@TableId

AND O.type='U'

AND OBJECT\_SCHEMA\_NAME(O.object\_id) NOT IN ('dba','validate','CandidateKey','sys')

AND O.name NOT LIKE 'sysdiagram%'

),

FKQualifier AS

(SELECT object\_id

FROM FKCandidate

GROUP BY object\_id

HAVING COUNT(\*)=@PKFieldCount -- Identify the table objects that have all the fields in the primary key.

),

PKFieldOrder AS (

SELECT StandardisedColumnName, ColumnID

FROM CandidateKey.StandardisedPKItems

WHERE Object\_Id = @TableID

)

INSERT INTO @FKTable(Object\_Id,ColumnID,StandardisedObjectName,OriginalColumnName,StandardisedColumnName)

SELECT

FKCandidate.object\_id,

PKFieldOrder.ColumnID,

FKCandidate.StandardisedObjectName,

FKCandidate.name,

FKCandidate.StandardisedColumnName

FROM FKQualifier INNER JOIN FKCandidate

ON FKQualifier.object\_id = FKCandidate.object\_id

INNER JOIN PKFieldOrder

ON PKFieldOrder.StandardisedColumnName = FKCandidate.StandardisedColumnName

ORDER BY PKFieldOrder.ColumnId -- Essential for multi-field relationships

The common table expressions used are as follows:

| **CTE** | **Description** |
| --- | --- |
| FKCandidate | Tables and fields where the fields match against any that appear in the parent primary key |
| FKQualifier | For those tables and fields identified in FKCandidate identify those tables that contain all the fields that appear in the parent primary key. |

Generating the full ALTER TABLE statement

This is simply a case of looping the records in.

DECLARE @NextObject INT=0 -- Child table object\_id

DECLARE @ForeignKeyText VARCHAR(MAX) -- Full blown ALTER TABLE ... FOREIGN KEY... REFERENCES statement.

DECLARE @ForeignKeyFields VARCHAR(MAX) -- delimited list of foreign key fields

WHILE @NextObject IS NOT NULL

BEGIN

SELECT @NextObject = MIN(Object\_Id)

FROM @FKTable

WHERE Object\_Id>@NextObject

IF @NextObject IS NOT NULL

BEGIN

IF NOT EXISTS(

SELECT FK.StandardisedColumnName

FROM @FKTable AS FK

INNER JOIN CandidateKey.GetFieldsUsedInExistingFK(@NextObject) AS CK

ON FK.StandardisedColumnName = CK.StandardisedColumnName

)

BEGIN

SET @ForeignKeyText='ALTER TABLE '

+ OBJECT\_SCHEMA\_NAME(@NextObject)+'.'+OBJECT\_NAME(@NextObject)

+ ' WITH NOCHECK ADD CONSTRAINT FKCandidate\_'+OBJECT\_SCHEMA\_NAME(@NextObject)+'\_'+OBJECT\_NAME(@NextObject)

+ '\_' +OBJECT\_SCHEMA\_NAME(@TableId)+'\_'+OBJECT\_NAME(@TableId)

+ ' FOREIGN KEY ('

-- Build concatenated list of foreign key names

SELECT @ForeignKeyFields = COALESCE(@ForeignKeyFields+',','')+OriginalColumnName

FROM @FKTable

WHERE Object\_Id=@NextObject

ORDER BY ColumnID

SET @ForeignKeyText=@ForeignKeyText

+ @ForeignKeyFields + ') '

+ @SQLReferences

PRINT @ForeignKeyText

BEGIN TRY

EXEC(@ForeignKeyText)

END TRY

BEGIN CATCH

PRINT '================================='

RAISERROR('ERROR TableID=%i, SQL=%s',10,1,@TableID,@ForeignKeyText) WITH NOWAIT

PRINT '================================='

END CATCH

SET @ForeignKeyFields = NULL

END

END

END

GO

Dealing with Multi-field relationships

This is simply a case of iterating through the list of objects with multiple field primary keys in reverse order of the number of fields.

For each object we simply call CandidateKey.GetCandidateForeignKey.

CREATE PROC CandidateKey.GenerateMultiFieldFK

AS

SET NOCOUNT ON

DECLARE @ProcName VARCHAR(256)=OBJECT\_SCHEMA\_NAME(@@PROCID)+'.'+OBJECT\_NAME(@@PROCID)

RAISERROR('PROC: %s',10,1,@ProcName) WITH NOWAIT

DECLARE

@object\_id INT,

@PKFieldCount TINYINT

DECLARE csr\_FK INSENSITIVE CURSOR

FOR SELECT

Object\_Id,

PKFieldCount

FROM CandidateKey.PKFieldCount

WHERE PKFieldCount>1

ORDER BY PKFieldCount DESC

OPEN csr\_FK

FETCH NEXT FROM csr\_FK INTO @object\_id,@PKFieldCount

WHILE @@FETCH\_STATUS = 0

BEGIN

EXEC CandidateKey.GetCandidateForeignKey @object\_id

FETCH NEXT FROM csr\_FK INTO @object\_id,@PKFieldCount

END

CLOSE csr\_FK

DEALLOCATE csr\_FK

GO

Relationships constrained to a schema

We have identified a rules based hierarchy that allows us to cater for the relationship between BusinessEntity, Person and Password in the Person schema within the Adventureworks2014 database.

The puzzle we now face is where we have BusinessEntityID as the primary key in the other schemas.  What we want to achieve is have objects within a schema that contain the BusinessEntityID field to relate to the table in their schema with BusinessEntityId as the primary key field and not link back to the Person.BusinessEntity object.

We want a stored procedure that generates a foreign key based on the following rules:

* The parent table must contain a single field primary key
* Both parent and child must exist in the same schema
* The child must not already have a parent relationship using the same fieldname as the primary key
* The parent table name must be at the start of the table name for the child.  That is Sales.SalesPerson is the parent of Sales.SalesPersonQuota.

The stored procedure to achieve this is shown below.

CREATE PROC CandidateKey.GenerateSchemaSpecificFK

AS

SET NOCOUNT ON

DECLARE @ProcName VARCHAR(256)=OBJECT\_SCHEMA\_NAME(@@PROCID)+'.'+OBJECT\_NAME(@@PROCID)

PRINT '==============================================='

RAISERROR('PROC: %s',10,1,@ProcName) WITH NOWAIT

DECLARE

@SQL VARCHAR(MAX),

@CRLF CHAR(2)=CHAR(13)+CHAR(10)

SELECT @SQL=COALESCE(@SQL+';'+@CRLF,'')

+ 'ALTER TABLE '

+ QUOTENAME(c.SchemaName)+'.'+QUOTENAME(C.OriginalObjectName)

+ ' WITH NOCHECK ADD CONSTRAINT '

+ QUOTENAME('FKCandidate\_'+c.SchemaName+'\_'+C.OriginalObjectName+'\_'+P.schemaName+'\_'+P.OriginalObjectname)

+ ' FOREIGN KEY ('

+ C.originalColumnName

+ ') REFERENCES '

+ QUOTENAME(P.SchemaName)+'.'+QUOTENAME(P.OriginalObjectName)

+ '('

+ p.OriginalColumnName

+ ')'

FROM CandidateKey.StandardisedPKItems AS P

INNER JOIN CandidateKey.PKFieldCount AS FC

ON P.Object\_Id = FC.Object\_Id

AND FC.PKFieldCount=1

INNER JOIN CandidateKey.StandardisedPKItems AS C

ON P.SchemaName = C.SchemaName

AND P.Object\_Id <> C.Object\_Id

AND PATINDEX(P.StandardisedObjectName+'%',C.StandardisedObjectName)=1

AND P.StandardisedColumnName = C.StandardisedColumnName

WHERE C.StandardisedColumnName NOT IN(SELECT StandardisedColumnName FROM CandidateKey.GetFieldsUsedInExistingFK( C.Object\_id ))

PRINT @SQL

EXEC(@SQL)

GO

Final sweep up of single field primary key relationships

So far we have dealt with the following items

1. One to 0..1 hierarchies within a schema
2. Multi-field relationships
3. One to many relationships within a schema

We now have to deal with cross schema relationships and any other relationships not catered for by the rules applied so far.

The puzzle to solve this time is that where there are 1 to 0..1 relationships we want to exclude any but the root object in that particular hierarchy.

Once we have a list of qualifying objects then we can simply iterate through that list of objects calling our CandidateKey.GetCandidateForeignKey stored procedure. The stored procedure below illustrates how this can be achieved.

CREATE PROC CandidateKey.GenerateFinalSingleFieldFK

AS

SET NOCOUNT ON

DECLARE @ProcName VARCHAR(256)=OBJECT\_SCHEMA\_NAME(@@PROCID)+'.'+OBJECT\_NAME(@@PROCID)

PRINT '==============================================='

RAISERROR('PROC: %s',10,1,@ProcName) WITH NOWAIT

DECLARE @SingleFieldPKs TABLE (Object\_Id INT NOT NULL PRIMARY KEY CLUSTERED)

DECLARE @Object\_Id INT=0

;WITH MO AS ( -- For the 1 to 0..1 relationships rank the priority

SELECT

Object\_id,

SequenceNumber=row\_number() OVER (PARTITION BY StandardisedColumnName ORDER BY StandardisedColumnName, TablePriority),

StandardisedColumnName

FROM CandidateKey.PKOneToOneZero

),

PriorityObject AS ( -- Choose the master table in the 1 to 0..1 relationships

SELECT StandardisedColumnName,Object\_Id

FROM MO

WHERE SequenceNumber=1

)

-- Exclude tables if they participate in a 1 to 0..1 relationship and are not the master.

INSERT INTO @SingleFieldPKs(Object\_Id)

SELECT FC.Object\_Id

FROM CandidateKey.PKFieldCount FC

INNER JOIN CandidateKey.StandardisedPKItemS AS SPK

ON FC.Object\_Id = SPK.Object\_Id

LEFT JOIN PriorityObject

ON SPK.StandardisedColumnName = PriorityObject.StandardisedColumnName

WHERE FC.PKFieldCount=1

AND FC.Object\_Id = ISNULL(PriorityObject.Object\_Id,FC.Object\_ID)

WHILE @Object\_Id IS NOT NULL

BEGIN

SELECT @Object\_Id = MIN(Object\_Id)

FROM @SingleFieldPKs

WHERE Object\_Id>@Object\_Id

IF @Object\_Id IS NOT NULL

EXEC CandidateKey.GetCandidateForeignKey @object\_id

END

GO

Putting the foreign key procs together

Running the process

This is as simple as running the following:-

exec CandidateKey.GetStandardisedPKItems;

exec CandidateKey.GetPKOneToOneZeroPriority;

exec CandidateKey.GeneratePKOneToOneZeroHierarchy;

exec CandidateKey.GenerateMultiFieldFK;

exec CandidateKey.GenerateSchemaSpecificFK;

exec CandidateKey.GenerateFinalSingleFieldFK;

Examining the results

My starting point was my copy of AdventureWorks2014.  From this I took a backup and restored a copy into a database called AW2014.

I destroyed all foreign keys in AW2014 and ran my process.

A straight count of the sysreferences table in both databases produced the following results

| **DatabaseName** | **NumberOfRelationships** |
| --- | --- |
| Adventureworks2014 | 92 |
| Aw2014 | 79 |

The numbers do not reveal the full story.  The following query highlights 37 different table to table relationships.

SELECT DISTINCT

OriginalParentName=Original.ParentName,

OriginalChildName=Original.ChildName,

NewParentName=NewR.ParentName,

NewChildName=NewR.ChildName

FROM

(

SELECT

ParentName=PS.name+'.'+P.name,

ChildName=CS.name+'.'+C.name

FROM Adventureworks2014..sysreferences AS R

INNER JOIN Adventureworks2014.sys.objects AS P

ON R.rkeyId = P.object\_id

INNER JOIN Adventureworks2014.sys.schemas AS PS

ON P.schema\_id = PS.schema\_id

INNER JOIN Adventureworks2014.sys.objects AS C

ON R.fkeyId = C.object\_id

INNER JOIN Adventureworks2014.sys.schemas AS CS

ON C.schema\_id = CS.schema\_id

) AS Original

FULL OUTER JOIN(

SELECCT

ParentName=PS.name+'.'+P.name,

ChildName=CS.name+'.'+C.name

FROM AW2014..sysreferences AS R

INNER JOIN AW2014.sys.objects AS P

ON R.rkeyId = P.object\_id

INNER JOIN AW2014.sys.schemas AS PS

ON P.schema\_id = PS.schema\_id

INNER JOIN AW2014.sys.objects AS C

ON R.fkeyId = C.object\_id

INNER JOIN AW2014.sys.schemas AS CS

ON C.schema\_id = CS.schema\_id

) NewR

ON Original.ParentName = NewR.ParentName

AND Original.ChildName = NewR.ChildName

WHERE Original.ParentName IS NULL

OR NewR.ParentName IS NULL

One example of the gap is in the Sales.CurrencyRate table.  In Adventureworks2014 this is related to the Sales.Currency table with two relationships neither of which have been detected by our process.

* FromCurrencyCode
* ToCurrencyCode

The stored procedure below caters for this occurence:

CREATE PROC CandidateKey.PartialKeyNameFKs

AS

SET NOCOUNT ON

DECLARE @ProcName VARCHAR(256)=OBJECT\_SCHEMA\_NAME(@@PROCID)+'.'+OBJECT\_NAME(@@PROCID)

PRINT '==============================================='

RAISERROR('PROC: %s',10,1,@ProcName) WITH NOWAIT

DECLARE @SQL VARCHAR(MAX)

DECLARE @CRLF CHAR(2)=CHAR(13)+CHAR(10)

;WITH PKFields AS (

SELECT

SPK.Object\_id,

SPK.ColumnId,

SPK.StandardisedColumnName,

SPK.OriginalColumnName,

SPK.DataTypeID,

SPK.SchemaName,

SPK.OriginalObjectName

FROM CandidateKey.StandardisedPKItems AS SPK

INNER JOIN CandidateKey.PKFieldCount AS PKFC

ON SPK.Object\_id = PKFC.Object\_id

AND PKFC.PKFieldCount=1 -- We are only interested in single field PKs

WHERE StandardisedColumnName NOT IN ('ID','Code') -- Exclude these as they generate huge numbers of false positives

AND DataTypeID NOT IN (SELECT system\_type\_id FROM sys.types WHERE name LIKE 'date%' OR name LIKE '%time%')

)

SELECT @SQL = COALESCE(@SQL+';'+@CRLF,'')

+ 'ALTER TABLE '

+ QUOTENAME(OBJECT\_SCHEMA\_NAME(O.object\_id))+'.'+QUOTENAME(O.name)

+ ' ADD CONSTRAINT '

+ QUOTENAME('FKCandidate\_'+OBJECT\_SCHEMA\_NAME(O.object\_id)+'\_'+O.name+'\_'+PKFields.SchemaName+'\_'+PKFields.OriginalObjectName+'\_'+C.Name)

+ ' FOREIGN KEY ('

+ C.Name

+ ') REFERENCES '

+ QUOTENAME(PKFields.SchemaName)+'.'+QUOTENAME(PKFields.OriginalObjectName)

+ '('

+ PKFields.OriginalColumnName

+ ')'

FROM sys.columns AS C

INNER JOIN sys.objects AS O

ON C.object\_id = O.object\_id

INNER JOIN CandidateKey.PKFieldCount AS PKFC

ON O.object\_id = PKFC.Object\_id

AND PKFC.PKFieldCount=1

INNER JOIN PKFields

ON C.system\_type\_id = PKFields.DataTypeID -- A FK must share the same data type from parent to child.

AND (

REPLACE(C.name,'\_','') LIKE PKFields.StandardisedColumnName+'%'

OR REPLACE(C.name,'\_','') LIKE '%'+PKFields.StandardisedColumnName

)

LEFT JOIN sys.sysreferences AS REF

ON REF.rkeyid = PKFields.Object\_id

AND REF.rkey1 = PKFields.ColumnId

AND REF.fkey1 = C.column\_id

AND REF.fkeyid = O.object\_id

WHERE O.type='U'

AND REPLACE(C.name,'\_','') <>PKFields.StandardisedColumnName

AND REF.rkeyid IS NULL

PRINT @SQL

EXEC(@SQL)

GO

However an example we cannot cater for is in the Production.BillOfMaterials table. In Adventureworks2014 this has three relationships of which AW2014 has one.  The two missing ones are as follows:

* BillOfMaterials.ProductAssemblyID = Product.ProductID
* BillOfMaterials.ComponentID = Product.ProductID

Neither of the above could be detected without some form a thesaurus capability.

An example of a false positive is that our process defined a relationship between ProductListPriceHistory and ProductCostHistory.  Both have a primary key of ProductID and StartDate.

Further differences involves the 1 to 0..1 relationship for the BusinessEntityId field.  To take an example of Sales.SalesPerson

* Adventureworks2014 the parent is HumanResources.Employee
* AW2014 the parent is Person.BusinessEntity

In all the cases mentioned above the reasons for the discrepancy can be explained by the limitations of a naming convention.

Concluding thoughts

The problem of auto-suggesting foreign key relationships is far more difficult than it would first appear.

Strong naming conventions help but are by no means a panacea.  If a database has strong naming conventions that suggests discipline and I would question whether a database produced using disciplined approaches would be likely to omit foreign key constraints.

I did consider a solution that involves parsing views, functions and stored procedures and even execution plans but rejected for the following reasons:

* There is no guarantee of views, functions and procs being present in the database
* A quick look at the objects within Adventureworks2014 suggests that parsing such objects is even more complicated that the naming convention solution.

I should be interested to hear how well the techniques and code described here work with real world examples.

Resources:

[ForeignKeyInvestigation.zip](http://www.sqlservercentral.com/Files/ForeignKeyInvestigation.zip/28040.zip)