**Menu**

Top of Form

Bottom of Form

**Automatically guessing foreign key constraints**

[2017-04-102017-04-10](https://sqlsunday.com/2017/04/10/guessing-fk-constraints/) / [Daniel Hutmacher](https://sqlsunday.com/author/dhmacher/)

FROM: <https://sqlsunday.com/2017/04/10/guessing-fk-constraints/?utm_source=Brent+Ozar+Unlimited%C2%AE+List&utm_campaign=e9491fcb77-Weekly_Links_2017_04_09&utm_medium=email&utm_term=0_8e3e861dd9-e9491fcb77-296071565&goal=0_8e3e861dd9-e9491fcb77-296071565&mc_cid=e9491fcb77&mc_eid=b137e91a2b>

With good naming and datatyping conventions, an automated script can help you with the process of creating foreign key constraints across your database, or actually, *suggest* table relations where you’ve forgotten to implement them.

However, the whole idea of automating foreign key constraint creation hinges on a few critical factors, which is why I’ve used the word “guessing” in the title of this post.

* The exact definition of foreign key constraints comes down to domain knowledge of your database – sometimes, what looks like a foreign key candidate really isn’t one.
* Fake NULLs: Do you use “n/a”, “0”, “-1” or “1900-01-01” when you actually mean NULL?
* Legacy code is full of exceptions to good modelling practices. Outside the english-speaking world, for instance, we frequently see a mix of languages, depending on what developer was working on this particular piece of code.
* This script won’t guess relations where the column name and datatype don’t match. For instance, there’s no magic way to match ManagerID to EmployeeID unless you try every row of every column in the database.

With those things aside, let’s take a look at creating a script.

**DMVs we’ll need**

SQL Server comes with a huge number of DMVs (**dynamic management views**, nothing to do with your driver’s licence). The dynamic management views we’ll be looking at today reflect the metadata of the database, i.e. tables, columns, datatypes, indexes and primary keys. Here are the main players:

**sys.objects and sys.tables**

The **sys.objects** DMV contains a list of each database object, including tables, views, stored procedures, sequences, primary keys, and so on. The **type** column describes what type of object it is, “U” for instance being a user table, which is what we’ll be looking for today’s purpose. **sys.tables** has the same columns as sys.objects, but only contains user table objects and contains a few extra columns that are specific only to tables.

**sys.schemas**

Like it says on the tin, this DMV contains a list of database schemas. Connects to sys.objects and sys.tables via the schema\_id column.

**sys.columns**

A list of the columns in each table, view, table value function, etc. The primary key here is (object\_id, column\_id) and it connects to sys.objects using the object\_id column.

**sys.types**

Contains datatypes and joins to sys.columns via the user\_type\_id column. We won’t actually be needing this, but I’m mentioning it just for good measure.

**sys.indexes and sys.index\_columns**

**sys.indexes** contains indexes, primary keys and unique constraints (which, apart from the SQL syntax, are all the same physical thing under the hood). The is\_unique column tells us if this is a unique index or constraint, which is interesting when we’re looking for primary keys. The primary key of sys.indexes is (object\_id, index\_id). We’re also going to be looking at the *kind* of index we’re dealing with: filtering indexes on type IN (1, 2) gets us just the clustered and non-clustered ones, not heaps, columnstore, spatial, xml indexes, etc.

**sys.index\_columns** contains the keys of each index. It connects to sys.indexes using (object\_id, index\_id) and its primary key is (object\_id, index\_id, column\_id), where (object\_id, column\_id) refers to sys.columns. The **key\_ordinal** column is the “ordering number” of the index key column, where 1 is the first column in the index, 2 is the second, and so on. Note that key\_ordinal=0 means that this column is only [INCLUDE](https://sqlsunday.com/2013/02/24/covering-indexes/)d, not actually indexed.

**Putting it together**

Here’s how they all fit together. First off, all the schemas, tables and columns:

SELECT s.[name] AS [Schema],

       t.[name] AS [Table],

       c.column\_id,

       c.[name] AS [Column],

       dt.[name] AS Datatype

FROM sys.schemas AS s

INNER JOIN sys.tables AS t ON s.[schema\_id]=t.[schema\_id]

INNER JOIN sys.columns AS c ON t.[object\_id]=c.[object\_id]

INNER JOIN sys.types AS dt ON c.user\_type\_id=dt.user\_type\_id

ORDER BY s.[name], t.[name], c.column\_id;

And here are all the indexes along with their key columns:

SELECT s.[name] AS [Schema],

       t.[name] AS [Table],

       i.index\_id,

       i.[type\_desc],

       i.[name] AS [Index],

--- [String concatenation trick](https://sqlsunday.com/2014/02/09/aggregate-string-concatenations/)

       SUBSTRING(CAST(

       (SELECT ', '+c.[name]

        FROM sys.index\_columns AS ic

        INNER JOIN sys.columns AS c ON ic.[object\_id]=c.[object\_id] AND ic.column\_id=c.column\_id

        WHERE ic.[object\_id]=i.[object\_id] AND ic.index\_id=i.index\_id AND ic.key\_ordinal>0

        ORDER BY ic.key\_ordinal

        FOR XML PATH(''), TYPE) AS varchar(4000)), 3, 4000) AS [Columns],

       SUBSTRING(CAST(

       (SELECT ', '+c.[name]

        FROM sys.index\_columns AS ic

        INNER JOIN sys.columns AS c ON ic.[object\_id]=c.[object\_id] AND ic.column\_id=c.column\_id

        WHERE ic.[object\_id]=i.[object\_id] AND ic.index\_id=i.index\_id AND ic.key\_ordinal=0

        FOR XML PATH(''), TYPE) AS varchar(4000)), 3, 4000) AS [Include]

FROM sys.schemas AS s

INNER JOIN sys.tables AS t ON s.[schema\_id]=t.[schema\_id]

INNER JOIN sys.indexes AS i ON t.[object\_id]=i.[object\_id] AND i.[type] IN (1, 2)

ORDER BY s.[name], t.[name], i.index\_id;

**Identifying referenced tables**

In a foreign key constraint, the two tables involved are the **referencing** and the **referenced**. The referencing table is the one to which you’ve attached the constraint, while the referenced table is the one to which the foreign key references. In this example,

ALTER TABLE dbo.Cars ADD

CONSTRAINT FK\_Cars\_Color (ColorID)

REFERENCES dbo.Colors (ColorID);

… the table dbo.Cars is the *referencing* table and dbo.Colors is the *referenced* table. You can tell because we reference dbo.Colors by its primary/unique key (ColorID), whereas ColorID is (hopefully) not the primary key of dbo.Cars.

Let’s create a table of possible referenceable tables, i.e. of all the unique indexes in the database:

DECLARE @referenced TABLE (

    [object\_id]        int NOT NULL,

    index\_id        int NOT NULL,

    column\_name        sysname NOT NULL,

    user\_type\_id    int NOT NULL,

    PRIMARY KEY CLUSTERED ([object\_id], index\_id, column\_name)

);

INSERT INTO @referenced ([object\_id], index\_id, column\_name, user\_type\_id)

SELECT t.[object\_id], i.index\_id, c.[name] AS column\_name, c.user\_type\_id

FROM sys.tables AS t

INNER JOIN sys.indexes AS i ON t.[object\_id]=i.[object\_id]

INNER JOIN sys.index\_columns AS ic ON i.[object\_id]=ic.[object\_id] AND i.index\_id=ic.index\_id

INNER JOIN sys.columns AS c ON ic.[object\_id]=c.[object\_id] AND ic.column\_id=c.column\_id

WHERE c.is\_nullable=0 AND i.[type] IN (1, 2) AND i.is\_unique=1 AND ic.key\_ordinal>0;

Rather than saving the fully qualified schema and table names, we’ll just save the object\_id, which uniquely identifies every object in the database.

At first glance, finding referencing objects sounds simple enough: just match the column names and datatypes, right?

SELECT t.[object\_id] AS referencing\_tbl, c.[name] AS referencing\_col,

       r.[object\_id] AS referenced\_tbl,  r.column\_name AS referenced\_col

FROM @referenced AS r

INNER JOIN sys.tables AS t ON

    r.[object\_id]!=t.[object\_id]

INNER JOIN sys.columns AS c ON

    t.[object\_id]=c.[object\_id] AND

    c.[name]=r.column\_name AND

    c.user\_type\_id=r.user\_type\_id;

But this won’t work when you have a *composite* primary key on the referenced table, meaning that you have two or more columns that make up the primary key together. The problem is that you can’t make sure if *all* the columns match, or if you’ve only matched some of them.

To make sure that we can match all of the columns, we’re going to break out our CTE skills and count the columns on the referenced tables, as well as the number of columns that we’ve been able to match. So if your primary key on the referenced table is a composite key of three columns, you should be able to join on all of those three columns in order to be able to create a foreign key constraint. If you’ve joined two out of three columns, this isn’t a valid foreign key constraint.

Our CTE happens in three steps. First, the referenced tables, to which we’re adding a count of the key columns:

WITH referenced AS (

    SELECT \*, COUNT(\*) OVER (PARTITION BY [object\_id], index\_id) AS col\_count

    FROM @referenced),

The col\_count column returns the number of keys/columns in the primary key or unique index. If you want to understand the OVER () syntax, read up on [windowed functions](https://sqlsunday.com/2013/03/31/windowed-functions/). Next, we’ll join the referencing and referenced sides like we did before, but we’ll also add a column count after the join:

     work AS (

    SELECT COUNT(\*) OVER (PARTITION BY r.[object\_id], r.index\_id, t.[object\_id]) AS referencing\_count,

           r.col\_count AS referenced\_count, r.index\_id,

           t.[object\_id] AS referencing\_tbl, c.[name] AS referencing\_col,

           r.[object\_id] AS referenced\_tbl,  r.column\_name AS referenced\_col

    FROM referenced AS r

    INNER JOIN sys.tables AS t ON

        r.[object\_id]!=t.[object\_id]

    INNER JOIN sys.columns AS c ON

        t.[object\_id]=c.[object\_id] AND

--- This is where the column naming logic

--- can be customized:

        c.[name]=r.column\_name AND

        c.user\_type\_id=r.user\_type\_id)

Finally, return the results, but only where @referenced.col\_count is equal to work.referencing\_count:

SELECT \*

FROM work

WHERE referencing\_count=referenced\_count

ORDER BY referencing\_tbl, referenced\_tbl;

**Prettifying the output**

Finally, we’ll clean up the output by writing out the full syntax to build the constraint, as well as checking if there’s already an existing foreign key constraint that covers each relation that we’ve identified.

The end result looks like this:

DECLARE @referenced TABLE (

    [object\_id]        int NOT NULL,

    index\_id        int NOT NULL,

    column\_name        sysname NOT NULL,

    user\_type\_id    int NOT NULL,

    PRIMARY KEY CLUSTERED ([object\_id], index\_id, column\_name)

);

INSERT INTO @referenced ([object\_id], index\_id, column\_name, user\_type\_id)

SELECT t.[object\_id], i.index\_id, c.[name] AS column\_name, c.user\_type\_id

FROM sys.tables AS t

INNER JOIN sys.indexes AS i ON t.[object\_id]=i.[object\_id]

INNER JOIN sys.index\_columns AS ic ON i.[object\_id]=ic.[object\_id] AND i.index\_id=ic.index\_id

INNER JOIN sys.columns AS c ON ic.[object\_id]=c.[object\_id] AND ic.column\_id=c.column\_id

WHERE c.is\_nullable=0 AND i.[type] IN (1, 2) AND i.is\_unique=1 AND ic.key\_ordinal>0;

WITH referenced AS (

    SELECT \*, COUNT(\*) OVER (

PARTITION BY [object\_id], index\_id) AS col\_count

    FROM @referenced),

     work AS (

    SELECT COUNT(\*) OVER (

PARTITION BY r.[object\_id], r.index\_id, t.[object\_id]) AS referencing\_count,

           r.col\_count AS referenced\_count, r.index\_id,

           t.[object\_id] AS referencing\_tbl, c.[name] AS referencing\_col,

           r.[object\_id] AS referenced\_tbl,  r.column\_name AS referenced\_col

    FROM referenced AS r

    INNER JOIN sys.tables AS t ON

        r.[object\_id]!=t.[object\_id]

    INNER JOIN sys.columns AS c ON

        t.[object\_id]=c.[object\_id] AND

--- This is where the column naming logic

--- can be customized:

        c.[name]=r.column\_name AND

        c.user\_type\_id=r.user\_type\_id)

SELECT fk.[name] AS [Existing FK],

       'ALTER TABLE '+ts.[name]+'.'+t.[name]+

' ADD CONSTRAINT '+

           ISNULL(fk.[name], 'FK\_'+rs.[name]+'\_'+r.[name]+'\_'+ts.[name]+'\_'+t.[name])+

' FOREIGN KEY ('+x.referencing\_columns+')'+

' REFERENCES '+rs.[name]+'.'+r.[name]+' ('+x.referenced\_columns+')' AS Syntax

FROM work

INNER JOIN sys.tables AS r ON work.referenced\_tbl=r.[object\_id]

INNER JOIN sys.schemas AS rs ON r.[schema\_id]=rs.[schema\_id]

INNER JOIN sys.tables AS t ON work.referencing\_tbl=t.[object\_id]

INNER JOIN sys.schemas AS ts ON t.[schema\_id]=ts.[schema\_id]

LEFT JOIN sys.foreign\_keys AS fk ON

    work.referencing\_tbl=fk.parent\_object\_id AND

    work.referenced\_tbl=fk.referenced\_object\_id

CROSS APPLY (

    SELECT

        SUBSTRING(CAST((

            SELECT ', '+w.referencing\_col

            FROM work AS w

            WHERE w.referencing\_tbl=work.referencing\_tbl AND

                  w.referenced\_tbl=work.referenced\_tbl AND

                  w.index\_id=work.index\_id

            ORDER BY w.referencing\_col

            FOR XML PATH(''), TYPE) AS varchar(4000)), 3, 4000),

        SUBSTRING(CAST((

            SELECT ', '+w.referenced\_col

            FROM work AS w

            WHERE w.referencing\_tbl=work.referencing\_tbl AND

                  w.referenced\_tbl=work.referenced\_tbl AND

                  w.index\_id=work.index\_id

            ORDER BY w.referencing\_col

            FOR XML PATH(''), TYPE) AS varchar(4000)), 3, 4000)

    ) AS x(referencing\_columns, referenced\_columns)

WHERE work.referencing\_count=work.referenced\_count

GROUP BY ts.[name], t.[name], rs.[name], r.[name], x.referencing\_columns, x.referenced\_columns, fk.[name]

ORDER BY Syntax;

Now, before you go crazy with this stuff, remember, it’s not a magic bullet, but rather some automation help to save you some coding and to help you review your data model. The script doesn’t change the database, it only prints out its suggestions, and this is totally by design.

* For this to work, you’ll obviously need proper primary keys or unique indexes on your referenced tables.
* We’re working on the assumption that the referencing and referenced column names are the same. Go ahead and change the script to suit your naming standards (look for the comment in the CTE)
* The script has no domain knowledge of your database, some of the suggestions are probably going to be downright silly.

I specifically used the old [xml string concatenation trick](https://sqlsunday.com/2014/02/09/aggregate-string-concatenations/) for this to work on older SQL Server versions. If you’re on SQL Server 2016, by all means, you could refactor the script to use STRING\_AGG().

Let me know how this works for you and if there’s anything you did to improve on the solution!