Efficiently Reuse Gaps in an Identity Column

2

By **Peter Larsson** on **9 February 2010** | 13 Comments | Tags: Identity

This article will demonstrate an efficient way to reuse gaps in an identity column. Please note that this is something you normally shouldn't be bothered about in a well-designed database or application. However, there are circumstances where you are forced to do this.

Missing identity values occur for a number of reasons. The most common reasons are roll backed transactions, failed inserts and deletes. If your table has TINYINT as identity column, it is very easy to max out this value. Here is an example of how to do this.

```
-- Create sample table
CREATE TABLE    #Sample
                (
                    RowID TINYINT IDENTITY(0, 1),
                    j CHAR(1)
                )
GO

-- Insert original sample data
INSERT  #Sample
SELECT  'P'

SELECT  *
FROM    #Sample
GO

-- Insert some sample data and rollback them
BEGIN TRAN
INSERT  #Sample
SELECT  'Q'
ROLLBACK TRAN

BEGIN TRAN
INSERT  #Sample
SELECT  'L'
ROLLBACK TRAN

SELECT  *
FROM    #Sample

-- Insert additional sample data
INSERT  #Sample
SELECT  'P'

SELECT  *
FROM    #Sample
GO

-- Try insert 200 failing sample data
INSERT  #Sample
SELECT  'PP'
GO 200

-- Insert additional sample data
INSERT  #Sample
SELECT  'P'

SELECT  *
FROM    #Sample
GO

-- Insert 45 additional sample data and remove them
INSERT  #Sample
SELECT  'Q'
GO 45
```


Resources

SQL Server Resources

Advertise on SQLTeam.com

SQL Server Books

SQLTeam.com Newsletter

Contact Us

About the Site

```
DELETE
FROM    #Sample
WHERE   j = 'Q'

SELECT  *
FROM    #Sample
GO

-- Insert additional sample data
INSERT  #Sample
SELECT  'P'

SELECT  *
FROM    #Sample
GO

-- Clean up
DROP TABLE #Sample
GO
```

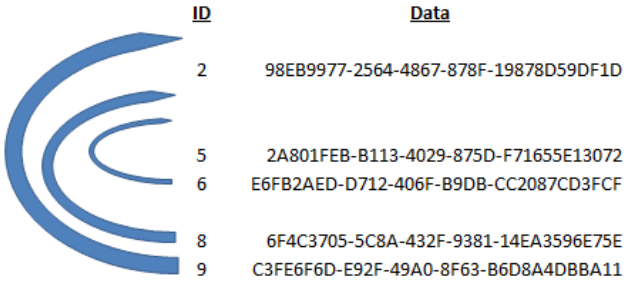
After this example, you can see that there are only four possible values left! We are now dangerously close to breaking the application. When the maximum value has been reached there is nothing you can do except changing the data type to a larger data type such as SMALLINT. But that might break schemas and you also may need to rewrite a number of stored procedures and rebuilding indexes.

In my opinion, this is a task that should be included in a DBA's job description: getting control over your identity values.

How Do We Fix This Scenario?

I first did some research to see which was the most common method to deal with this situation and not surprisingly the method of iterating all records from start to end was used! This method is not efficient. What if you have 1 million records and there is only 1 gap at position 15? Do you really want to shuffle all 1 million records to be sure to get rid of the gap? The answer is of course NO.

Having this in mind, I developed an algorithm which I call a folding algorithm. It defaults to reassign identity values at 1.



ID	Data
2	98EB9977-2564-4867-878F-19878D59DF1D
5	2A801FEB-B113-4029-875D-F71655E13072
6	E6FB2AED-D712-406F-B9DB-CC2087CD3FCF
8	6F4C3705-5C8A-432F-9381-14EA3596E75E
9	C3FE6F6D-E92F-49A0-8F63-B6D8A4DBBA11

The algorithm looks for how many missing ID's there are and pick the same number of records from the end of list. For the one million table example above, since there is only one ID missing, I only have to shuffle one record from the end of table to the missing position. Efficient and clean because only one record has to be touched and not the other 999 986 records!

Since an IDENTITY column most often also is used as a primary key, I have included one auxiliary table which references the base table to fix. In a production environment, this can be several tables, but I only include one for demonstration purposes.

Begin with creating some sample data like this:

```
--  Create sample table
CREATE TABLE    #Sample
            (
                ID INT IDENTITY(1, 1) PRIMARY KEY CLUSTERED,
                Data UNIQUEIDENTIFIER DEFAULT NEWID()
            )
GO

-- Populate sample data
INSERT  #Sample
DEFAULT VALUES
GO 10

-- Remove some records to mimic real world scenario
DELETE
FROM    #Sample
WHERE   ID IN (1, 3, 4, 7, 10)
```

```
-- Check content of sample table
SELECT  *
FROM    #Sample

-- Create auxiliary table for foreign key issues
CREATE TABLE    #Aux
                (
                    AuxID INT IDENTITY(1, 1) PRIMARY KEY CLUSTERED,
                    SampleID INT NOT NULL REFERENCES #Sample(ID)
                )

INSERT  #Aux
SELECT  9 UNION ALL
SELECT  9 UNION ALL
SELECT  8

-- Check content of auxiliary table
SELECT  *
FROM    #Aux
```

Now we have created a sample table with 10 records and then on purpose created some gaps by removing some records manually. We also created an auxiliary table referencing the base table. Since foreign key constraints are not enforced on temporary tables, this is for demonstration only.

The first step in the algorithm is to create the translation table. This table will hold all the missing ID's and which ID to shuffle. This piece of code is the heart of the algorithm. It will seek out all missing ID's, compare them to how many records there are in total, and "fold" them together. What the algorithm does, is to calculate the missing ID's and pick the appropriate number of records from end of table to shuffle and fill the gaps.

```
-- Set up ID translation with Peso's "folding" algorithm
CREATE TABLE    #Trans
                (
                    idNew INT NOT NULL,
                    idOld INT NOT NULL
                )

INSERT      #Trans
            (
                idNew,
                idOld
            )
SELECT      d.rn,
            w.ID
FROM        (
                SELECT      d.rn,
                            ROW_NUMBER() OVER (ORDER BY d.rn) AS recID
                FROM        (
                                SELECT  ROW_NUMBER() OVER (ORDER BY ID) AS rn
                                FROM    #Sample
                            ) AS d
                LEFT JOIN   #Sample AS s ON s.ID = d.rn
                WHERE       s.ID IS NULL
            ) AS d
INNER JOIN  (
                SELECT  ID,
                        ROW_NUMBER() OVER (ORDER BY ID DESC) AS recID
                 FROM   #Sample
            ) AS w ON w.recID = d.recID
```

With this in place we can now begin to create duplicate entries of the records to shuffle. This is risk free since new inserts are based on the current identity value so these forced inserted records will not collide.

Now when we have the #Trans control table, let's begin to shuffle the records and update auxiliary table(s).

```
-- Prepare source table for identity inserts
BEGIN TRAN

SET IDENTITY_INSERT #Sample ON

INSERT        #Sample
              (
                  ID,
                  data
              )
SELECT      t.idNew,
            s.data
```

```
FROM        #Trans AS t
INNER JOIN  #Sample AS s ON s.ID = t.idOld

IF @@ERROR <> 0
    BEGIN
        ROLLBACK TRAN
        RAISERROR('Insert old records with new ID failed.', 18, 1)
        RETURN
    END

SET IDENTITY_INSERT #Sample OFF

UPDATE      w
SET         w.SampleID = t.idNew
FROM        #Aux AS w
INNER JOIN  #Trans AS t ON t.idOld = w.SampleID

IF @@ERROR <> 0
    BEGIN
        ROLLBACK TRAN
        RAISERROR('Update #Aux table failed.', 18, 1)
        RETURN
    END

DELETE      s
FROM        #Sample AS s
INNER JOIN  #Trans AS t ON t.idOld = s.ID

IF @@ERROR <> 0
    BEGIN
        ROLLBACK TRAN
        RAISERROR('Delete old records failed.', 18, 1)
        RETURN
    END

COMMIT TRAN
```

And now we're set, right? Well, not quite right. Since the IDENTITY value for the base table is stored in the Meta data for the table, we need to reset the current IDENTITY value so that we do not produce another gap at the end of IDENTITY sequence. You can do that like this

```
-- Cleanup
DECLARE @ID INT

SELECT  @ID = MAX(ID)
FROM    #Sample

DBCC CHECKIDENT(#Sample, RESEED, @ID)

DROP TABLE  #Sample,
            #Aux,
            #Trans
```

Remember to put this piece of code in the transaction as well, for continuity.

Conclusion

I have tested this technique in a production system with several users online, and for a reasonable amount of gaps you can run this piece of code during normal operation. But remember the base table will be locked for a small amount of time and it will slow down other processes accessing the base table. The best practice is to run this algorithm off-hours.

Good luck!

Related Articles

How to Insert Values into an Identity Column in SQL Server

Other Recent Forum Posts

(6 August 2007)

Custom Auto-Generated Sequences with SQL Server (24 April 2007)

Using the OUTPUT Clause to Capture Identity Values on Multi-Row Inserts (14 August 2006)

Understanding Identity Columns (9 March 2002)

Identity and Primary Keys (28 February 2001)

Alternatives to @@IDENTITY in SQL Server 2000 (19 September 2000)

Uniqueidentifier vs. IDENTITY (12 September 2000)

Returning @@IDENTITY back to an ASP Page (18 August 2000)

get filename from string (5 Replies)

Drillthrough from cube to ssrs report (3 Replies)

Upgrade from 2000 or before (6 Replies)

insert select (4 Replies)

SELECT status in 3,4 and Not in 5 (11 Replies)

linked servers (2 Replies)

PK & FK (2 Replies)

Month Key Convert (3 Replies)