Curing Data-Obesity in OLTP Databases

It is quite common to have an OLTP database that must store large amounts of data which pile up into hundreds of millions, even billions, of rows. What do we do in such cases? In this article I will describe a way to deal with constant flows of OLTP data into production systems, and how to offload this data, describing the process from beginning to end.

Here are some assumptions for the sake of this article:

- We have a 50M rows of data in a production database
- The table contains log data. There are 100,000 rows coming per day of data related to website visits)
- We are using SQL Server 2012 Enterprise edition
- We have two SQL Servers – one is for highly-available transactional databases and another one for staging data purposes

How do we tackle the problem?
This is an illustration of the classic problem of mixing OLTP and Analytics data. We have a database which was designed to handle OLTP requirements, but it ends up gathering data which is required for analytical purposes. As a general rule, OLTP databases should be light on their feet. They ought to be small, storing only very recent data that should fit in memory so we can easily do In-Memory-OLTP. (Please note that no fancy technology is needed for In-Memory-OLTP: anyone who is smart enough can do In-Memory-OLTP, since memory nowadays is very cheap and the amount of relevant transactional data is finally getting behind the available memory capacities).
There are several ways to deal with this situation:

- Ignore the problem until the data volume grows so much that it starts to jeopardize the database backups and restores, hence availability
- Copy some of the older data to a different location and delete it from the main database
- Leave a very small portion of the data in the production system and automate the rotation of the data and the copying of it to the secondary location

Neither of the first two options are much good, but the third one seems quite reasonable and we will explore it in this article.

The Solution

The setup:

For this article we can assume that we have a database called **ProdLogDB**, which has a table **dbo.Log,** which is designed like this:

```
CREATE TABLE [dbo].[LOG]

   (

     [column1] [VARCHAR](50) NULL ,

     [column2] [VARCHAR](50) NULL ,

     [column3] [VARCHAR](50) NULL ,

     [column4] [VARCHAR](100) NULL ,

     [notes] [VARCHAR](MAX) NULL ,

     [timestamp] [DATETIME] NOT NULL ,

     [log_id] [INT] IDENTITY(1, 1)

                    NOT NULL ,

     CONSTRAINT [PK_LOG] PRIMARY KEY CLUSTERED ( [log_id] ASC )

   )

ON  [PRIMARY]
```

Now we need lots of data, and good representative data too (you can generate some data with the help of any Data Generator). After the generation of data the **dbo.Log** table will have data for the time between 2012-01-01 and 2015-01-20. For this time frame, have 10 million rows and the data is 1.5Gb. Will be using the same data generation project to generate 100,000 rows for the next few days, while we do the database management and data distribution.

The plan of action:

We have the data, but what should we do next? The idea is to have only a very small portion of the data in our production database, and to offload the rest of the data to a secondary server where the historical data will be stored. In our case we need only 7 days of data in the production system, so we can do analysis: The rest of the data can be someplace else. Furthermore, we want to automate the offloading process in such a way that a scheduled job will run every day and will move the oldest day's data to the secondary location. (The code below is flexible enough to support the rotation of 1 month's data, and we can provide a parameter to specify how many days of data to keep, up to 31 days.)

How do we achieve that?

Firstly, we need to ‘empty’ the **dbo.Log** table and move the data to a table containing the historical data. There are many ways to do this: We can, for example, bulk-copy the data to a different table, and batch-delete most of the data from the primary table. This, of course, would work, but it will take a long time and use a lot of resources, even if we do batch deletes. I would prefer the following strategy:

- We Create a new table called dbo.Log1, which has identical schema as the current production table, but it is partitioned, and ready for data writes from the application.

  We run the following script to do this:

  ```
  CREATE PARTITION FUNCTION PF_RingRecyclerByMonth (TINYINT)

  AS RANGE LEFT FOR VALUES (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  ,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31)

  CREATE PARTITION SCHEME PS_RingRecyclerByMonth

  AS PARTITION PF_RingRecyclerByMonth ALL TO ([PRIMARY])

  CREATE TABLE [dbo].[LOG1]

      (

     [column1] [VARCHAR](50) NULL ,

     [column2] [VARCHAR](50) NULL ,

     [column3] [VARCHAR](50) NULL ,

     [column4] [VARCHAR](100) NULL ,

     [notes] [VARCHAR](MAX) NULL ,

     [timestamp] [DATETIME] NOT NULL ,

     [log_id] [INT] IDENTITY(11000000, 1)

              NOT NULL ,

     [Offset] AS ( CONVERT([TINYINT], DATEPART(DAY, [timestamp])) ) PERSISTED

    )

  ON  PS_RingRecyclerByMonth(Offset)

   GO
  ```

  An important point to note here is that we are creating a new table, with one additional persisted computed column called **[Offset]**. This column is used to partition the table by the day number.

We will use this functionality later on to find the oldest partitions and to send them to the historical table.

Also, it is worth noting that we have the identity column **[log_id],** however the identity seed starts from a certain level. This is very important to get right, since we do not want to have duplicate ids later on. Make sure to set the identity seed from a level higher than the current one, even if there is a small gap.

In my case I have 10,000,000 rows, and I am estimating that in the next few days I will have 10,300,000, so I will set the seed to be 11,000,000. There will be some gap but better to be on the safe side.

- So far, we have one empty partitioned table (**dbo.Log1**), and one constantly growing production table (**dbo.Log**). Now it is time to do something buzzingly exciting: we will move data from one table to the other. In the matter of a second. Here is how:

```
BEGIN TRANSACTION;

EXEC sp_rename 'dbo.LOG', 'LOG_History';

EXEC sp_rename 'dbo.LOG1', 'LOG';

COMMIT TRANSACTION;
```

  So here is what just happened: our application was writing to the **dbo.Log** table, which was not partitioned and now, after renaming the tables, the application is writing to the newly created partitioned table. If the application is smartly written, we won't need to schedule for downtime or maintenance window. The rename should take less than a second and the application should retry to write again any rows that fall in the time of the rename.

- At this point we have two tables – **dbo.Log** – this one is partitioned and growing, and **dbo.Log_History** – this one is our large table which is not written to anymore.
- Now it is time to offload the historical data to our secondary server. One way to do it is to create a new database on the secondary server and bulk-copy the data. Another way to do it is to simply backup the current production database and restore it as a historical database on the secondary server. Either way, the result will be the same.

  After the successful copy of the data to the secondary server we can now delete the large table from our production server. (some magic may have to be performed to regain the allocated space after removing the large table, but this is a trivial task for a DBA)

  Also, keep in mind that your DBA will need to do some wizardry if your secondary server is not an Enterprise edition, since the database won't restore if it contains enterprise edition objects in it.

- At this point we have two databases on two different servers, one is called **ProdLogDB** and contains the partitioned table **dbo.Log,** and the other is called **ProdLogDB_History** and contains the **dbo.Log_History** table.

How do we rotate the production table, though?

The idea is to take the oldest partitions and to move them to the historical database.

First we need to create an empty table with the same structure so we can do partition switching. Create the same table in both databases:

```
CREATE TABLE [dbo].[LOG_SwitchTarget]

      (

    [column1] [VARCHAR](50) NULL ,

    [column2] [VARCHAR](50) NULL ,

    [column3] [VARCHAR](50) NULL ,

    [column4] [VARCHAR](100) NULL ,

    [notes] [VARCHAR](MAX) NULL ,

    [timestamp] [DATETIME] NOT NULL ,

    [log_id] [INT] NOT NULL ,

    [Offset] AS ( CONVERT([TINYINT], DATEPART(DAY, [timestamp])) ) PERSISTED

  )

ON  PS_RingRecyclerByMonth(Offset)

GO
```

Note that there is no identity specification on the **[log_id]** conlumn.

After creating the tables, we will use them to switch the partitions to them by using the following stored procedure:

```
IF OBJECT_ID('dbo.RingbufferRotate_ByMonth') IS NULL

   EXEC ('CREATE PROCEDURE dbo.RingbufferRotate_ByMonth AS RETURN 0;')

GO



ALTER  PROCEDURE RingbufferRotate_ByMonth

   @Now DATETIME = NULL ,

   @PartitionsToKeep INT = 7

AS

   SET NOCOUNT ON
```

```
    BEGIN

        IF @Now IS NULL

            SET @Now = GETDATE()

        DECLARE @BufferSize INT

        SELECT  @BufferSize = COUNT(*)

        FROM    sys.partitions P

                JOIN sys.tables T ON P.object_id = T.object_id

        WHERE   T.name = 'LOG'

        IF @PartitionsToKeep > @BufferSize

            BEGIN

                RAISERROR ('Can''t keep more partitions than the current buffer size of: %i', 16, 1,
@BufferSize)

                RETURN

            END

        DECLARE @CurrentPartition INT

        SET @CurrentPartition = DATEPART(DAY, @Now)

        DECLARE @OldestPartitionToKeep INT

        SET @OldestPartitionToKeep = ( @CurrentPartition + @BufferSize

                                       - @PartitionsToKeep ) % @BufferSize
```

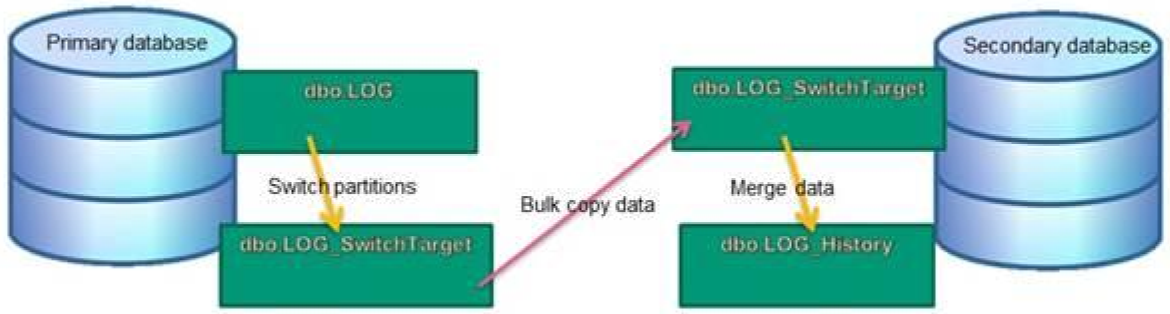
```
/* Start from the next partition up from current and move forward */

    DECLARE @P INT = ( @CurrentPartition + 1 ) % @BufferSize

    DECLARE @I INT = 0

    WHILE @I < @BufferSize - @PartitionsToKeep

      BEGIN

        DECLARE @Sql NVARCHAR(4000) = 'ALTER TABLE LOG SWITCH PARTITION <p>
TO LOG_SwitchTarget PARTITION <p>'

        SET @Sql = REPLACE(@Sql, '<p>', CAST(@P + 1 AS NVARCHAR))

        EXEC sp_executesql @Sql

        SET @I = @I + 1;

        SET @P = ( @P + 1 ) % @BufferSize

      END

  END
```

This procedure takes two parameters:

- **@Now**, which by default is set to get the current timestamp. It can be used to set the time to a different point and rotate the partitions from there
- **@PartitionsToKeep**, which is used to specify how many partitions (i.e. days of data) to keep in the current transactional table and how many to move to the **[dbo]**.**[LOG_SwitchTarget]** table. This value can be up to 30, but for our case let's choose to keep only 7 days of data and move all other data

At this point we don't have that much data in our Log table, since we renamed the tables fairly recently. But in a few days the data will pile up and our procedure will be useful.

Here is a picture of how the data transfer looks like:



Note that there are many different ways to implement this solution, but the important part is the automated partitions switching, the bulk-copying of the data from the **[dbo].[LOG_SwitchTarget]**

Table on the production server to the **[dbo].[LOG_SwitchTarget]** table on the secondary server and merging the data to the **[dbo].[LOG_History]** table.

Here is the **MERGE** procedure which should be created on the secondary server:

```
IF OBJECT_ID('dbo.Log_History_Merge') IS NULL

   EXEC ('CREATE PROCEDURE dbo.Log_History_Merge AS RETURN 0;')

GO

ALTER PROCEDURE dbo.Log_History_Merge

AS

   SET IDENTITY_INSERT LOG_History ON

   MERGE INTO LOG_History AS T

   USING

      ( SELECT   [column1] ,

                 [column2] ,

                 [column3] ,

                 [column4] ,

                 [notes] ,

                 [timestamp] ,

                 [log_id]

        FROM     [dbo].[LOG_SwitchTarget]

      ) AS A

   ON  A.[log_id] = T.[log_id]

       WHEN NOT MATCHED THEN

      INSERT ( [column1] ,

               [column2] ,

               [column3] ,
            [column4] ,

            [notes] ,

            [timestamp] ,

            [log_id]

           )

      VALUES ( A.[column1] ,

            A.[column2] ,

            A.[column3] ,

            A.[column4] ,

            A.[notes] ,

            A.[timestamp] ,

            A.[log_id]

           );

   SET IDENTITY_INSERT LOG_History OFF

   TRUNCATE TABLE [LOG_SwitchTarget];
```

From this point on it is very easy to implement the solution. For example, we can create an SSIS package which will run daily and will carry out the following tasks:

- Execute the **RingbufferRotate_ByMonth** procedure, which will switch all older partitions to the **LOG_SwitchTarget** table
- Bulk-copy the data from the **LOG_SwitchTarget** table on the primary server to the **LOG_SwitchTarget** table on the secondary server
- Execute the **Log_History_Merge** procedure on the secondary server
- Truncate the **LOG_SwitchTarget** tables on both servers

In order to test the code, let's get back to our test scenario: we have 10 million rows in the Log_History table. Let's generate some data in the LOG table, which will have timestamp between 2015-01-20 and 2015-01-28. We can then run the **RingbufferRotate_ByMonth** procedure and take a look at the data distribution per partition:

```
SELECT  $PARTITION.PF_RingRecyclerByMonth(OFFSET) AS PARTITION ,

      COUNT(*) AS [COUNT] ,

      CONVERT(DATE, [timestamp]) Datestamp
```

```
FROM   dbo.[LOG]

GROUP BY $PARTITION.PF_RingRecyclerByMonth(OFFSET) ,

        CONVERT(DATE, [timestamp])

ORDER BY CONVERT(DATE, [timestamp]);
```

This query returns data from the **LOG** table, which looks like this:

PARTITION	COUNT	Datestamp
23	111111	2015-01-23
24	111111	2015-01-24
25	111111	2015-01-25
26	111111	2015-01-26
27	111111	2015-01-27
28	111111	2015-01-28

And the following query will show us what we have in the **LOG_SwitchTarget** table:

```
SELECT  $PARTITION.PF_RingRecyclerByMonth(OFFSET) AS PARTITION ,

        COUNT(*) AS [COUNT] ,

        CONVERT(DATE, [timestamp]) Datestamp

FROM    dbo.[LOG_SwitchTarget]

GROUP BY $PARTITION.PF_RingRecyclerByMonth(OFFSET) ,

        CONVERT(DATE, [timestamp])

ORDER BY CONVERT(DATE, [timestamp]);
```

The data looks like this:

PARTITION	COUNT	Datestamp
20	111112	2015-01-20
21	111111	2015-01-21
22	111111	2015-01-22

Now it is time to do a bulk-copy of the data from the LOG_SwitchTarget table on the primary server to the LOG_SwitchTarget table on the secondary server.

After the bulk-copy has run, we can execute the dbo.Log_History_Merge procedure. Now we can see that we have the new data into our LOG_History table on the secondary server.