# Using partitioned tables or multiple tables to improve delete performance

Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:28 AM PST

Contents

**Using partitioned tables or multiple tables to improve the performance of big Deletes**

The content of this article was adapted from Thamires' original blog post.

## Issue

There are cases in which the time it takes to perform large Delete operations is higher than acceptable, even if there are no resource constraints and the execution plan is good. Azure SQL Database and Managed Instance only have a certain amount of throughput even on the highest SLO levels, and thus have an upper limit for such operations even if the Delete process is optimized. This limit might not be sufficient to meet business needs.

In such a cases, the following suggestions might improve the Delete operation.

## Investigation / Analysis

The main limitations on Delete operations usually are the throughput limits for Data I/O and Log I/O, and the size limit for the transaction log. On Premium and Business Critical databases, the replication lag between primary and secondary instances can introduce further limitations. Delete operations can also lead to high fragmentation in the tables, requiring further maintenance after the Delete was successful.

There are two options to avoid these limitations:

1. using multiple tables
2. using a partitioned table.

Either solution which will not only improve the Delete performance itself, but also reduce transaction log consumption, reduce table fragmentation and eliminate the need to scale up the database and reduce the costs, or prevent increasing it.

## Mitigation

The examples demonstrated in the Mitigation section are based on a simple log table with the following schema:

```
CREATE TABLE dbo.[Log]
(
    ID INT NOT NULL IDENTITY (1, 1),
    insert_time DATETIME NOT NULL DEFAULT GETDATE()
);
ALTER TABLE dbo.[Log] ADD CONSTRAINT [PK_Log_ID] PRIMARY KEY CLUSTERED (ID);
```

Column "insert_time" is a datetime that stores when the row was inserted. The only index present is the primary key PK_Log_ID. We will perform the clean-up task once a month and delete everything that is older than 2 months.

From a customer perspective: please keep in mind **you should implement your own process and test it in a non-production environment first.**

### Store the data round-robin by month in different tables

We could switch between 3 tables once a month and truncate the one that contains the older data. We can have a view with the `UNION ALL` of the 3 tables if necessary. The applications only write to the current table and read from either tables or view, but they cannot write to the view.

If the application only writes to the current table and users only read from the others manually when necessary, it may not be necessary to change the application. However, if the application needs to read and write the data, it may be necessary to make a few changes to have it read from the view or it would only show the most recent data (< 1 month). You need to take this in consideration when performing this change.

**Example**

```
-- Initial setup:
--
-- 1. Create two additional tables with the same schema to store the older data per month, as for example:
CREATE TABLE dbo.[Log_1month]
(
   ID INT NOT NULL IDENTITY (1, 1),
   insert_time DATETIME NOT NULL DEFAULT GETDATE()
);
ALTER TABLE dbo.[Log_1month] ADD CONSTRAINT [PK_Log_1month_ID] PRIMARY KEY CLUSTERED (ID);

CREATE TABLE dbo.[Log_2month]
(
   ID INT NOT NULL IDENTITY (1, 1),
   insert_time DATETIME NOT NULL DEFAULT GETDATE()
);
ALTER TABLE dbo.[Log_2month] ADD CONSTRAINT [PK_Log_2month_ID] PRIMARY KEY CLUSTERED (ID);

-- 2. Grant permissions to the new tables
--
--

-- 3. Create a new index for insert_time on the tables.
CREATE INDEX [Log_insert_time] ON [Log] (insert_time);
CREATE INDEX [Log_1month_insert_time] ON [Log_1month] (insert_time);
CREATE INDEX [Log_2month_insert_time] ON [Log_2month] (insert_time);

-- 4. Create a view to select the 3 tables if necessary, as for example:
CREATE VIEW dbo.[Log_select]
AS
    SELECT * FROM dbo.[Log]
    UNION ALL
    SELECT * FROM dbo.[Log_1month]
    UNION ALL
    select * from dbo.[Log_2month];

-- 5. Toggle between tables
DECLARE @minDate DATE;
DECLARE @limitDate DATE;

-- We will remove everything that is older than 2 months, so that is going to be the limitDate:
SET @limitDate = CAST(DATEADD(month, -2, CURRENT_TIMESTAMP) AS DATE);

-- Checking what is the newest data in the table log_2month:
SELECT @minDate = MAX(insert_time) FROM dbo.[Log_2month];
print @minDate
print @limitDate

-- If the table log_2month has data newer than 2 months, which is the retention period, the process is not exe
IF (@minDate <= @limitDate OR @minDate IS NULL)
BEGIN
    TRUNCATE TABLE dbo.[Log_2month]

    BEGIN TRANSACTION
    EXEC sp_rename 'dbo.Log_2month', 'Log_new';
    EXEC sp_rename 'dbo.Log_1month', 'Log_2month';
    EXEC sp_rename 'dbo.Log', 'Log_1month';
    EXEC sp_rename 'dbo.Log_new', 'Log';

        --Change the identity of the table to continue from the ID the other one was
    DECLARE @currentSeedValue INT;
    DECLARE @changeIdent_sql NVARCHAR(max);
    SET @currentSeedValue = IDENT_CURRENT('dbo.Log_1month');
    SET @changeIdent_sql = 'DBCC CHECKIDENT (''dbo.Log'', RESEED, ' + CAST(@currentSeedValue AS VARCHAR) + ')'
    EXEC sp_executesql @changeIdent_sql;
    COMMIT;
END
ELSE
```

```
        print 'No need to run this process, it might have been executed recently!'
```

## Partition the current table or create a new partitioned table

It is important to point out that, since on Azure SQL Database you do not have control on where the data is physically stored, we can only configure the partitions to use the primary filegroup.

If we partition by insert_time, keeping the primary key in the ID, we need to first recreate the primary key as nonclustered. It would also require to, every time we perform the clean-up process, drop the primary key before performing the truncate and recreate it afterwards, because it will not be a partitioned index, so it won't support this operation. In terms of efficiency, I believe this is not the best option, so I recommend partitioning by ID.

If we partition by the ID, it will not be as precise as the insert_time when we need to perform the truncate. We might have to leave a partition with data that has already reached the retention period, because it might also have newer data. The amount of data will depend on the range of each partition, so this can be mitigated by having smaller ranges.

We could partition the current table or create a new partitioned table: The creation of a new partitioned table would avoid the maintenance window, but it would require to grant permissions and rename the tables, so the new one can assume the place of the old one and the application can start writing to it. We could then keep the old table for the historical data until the retention period is reached.

### Example (partitioning the table)

```
-- Create the partitions for every 100.000 records, but you can reduce the range as much as you would like
CREATE PARTITION FUNCTION [PF_PARTITIONBYID](int) AS RANGE RIGHT
FOR VALUES ('100000','200000','300000','400000','500000','600000', '700000','800000','900000','1000000',
    '1100000','1200000','1300000','1400000','1500000', '1600000','1700000','1800000','1900000','2000000');

-- As it is not possible to manage where the data will be physically stored on Azure SQL DB, you have to set a
CREATE PARTITION SCHEME [PS_PARTITIONBYID]
AS PARTITION [PF_PARTITIONBYID]
ALL TO ([PRIMARY]);

-- You can recreate the PK as the clustered partitioned index
-- cross-check the actual constraint name if this fails:
ALTER TABLE dbo.[Log] DROP CONSTRAINT [PK_Log_ID];

ALTER TABLE dbo.[Log] ADD CONSTRAINT [PK_Log] PRIMARY KEY CLUSTERED  (ID)
    WITH (STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
    ON PS_PARTITIONBYID(ID);

-- You can create a partitioned index on insert_time and ID to make the search for the partitions to remove/tr
CREATE INDEX Log_insert_time ON dbo.[Log] (insert_time, ID);

--To check the partitions
SELECT SCHEMA_NAME(o.schema_id) + '.' + OBJECT_NAME(i.object_id) AS [object]
       , p.partition_number
       , i.index_id
       , p.rows
       , CASE boundary_value_on_right
             WHEN 1 THEN 'less than' ELSE 'less than or equal to'
         END AS comparison
       , rv.value
FROM sys.partitions p
INNER JOIN sys.indexes i ON p.object_id = i.object_id AND p.index_id = i.index_id
INNER JOIN sys.objects o ON p.object_id = o.object_id
INNER JOIN sys.partition_schemes ps ON ps.data_space_id = i.data_space_id
INNER JOIN sys.partition_functions f ON f.function_id = ps.function_id
INNER JOIN sys.destination_data_spaces dds ON dds.partition_scheme_id = ps.data_space_id AND dds.destination_i
INNER JOIN sys.filegroups fg ON dds.data_space_id = fg.data_space_id
LEFT OUTER JOIN sys.partition_range_values rv ON f.function_id = rv.function_id AND p.partition_number = rv.bo
WHERE o.object_id = OBJECT_ID('dbo.Log')
```

Note: For partition-switching with a separate, new table, it would be basically the same steps, since we first need to create a copy without data of the old one. The only thing that would change would be the name of the table in which we would be creating the partition and the rename of both tables at the end.

### Example (truncating the partition)

```
-- Get the list of partitions to be removed, based on the 2 months retention period
DECLARE @Max_id INT;
DECLARE @truncate_sql NVARCHAR(max)
DECLARE @merge_sql NVARCHAR(max)
SELECT @Max_id = MAX(ID) FROM dbo.[Log] WHERE insert_time < CAST(DATEADD(month, -2, CURRENT_TIMESTAMP) AS DATE

SELECT
     @truncate_sql = 'TRUNCATE TABLE dbo.[Log] WITH (PARTITIONS(' + CAST(MIN(partition_number) AS VARCHAR) + '
   , @merge_sql = 'ALTER PARTITION FUNCTION [PF_PARTITIONBYID]() MERGE RANGE (' + CAST(MAX(rv.value) AS VARCHA
FROM
    sys.partitions p
    INNER JOIN sys.indexes i ON p.object_id = i.object_id AND p.index_id = i.index_id
    INNER JOIN sys.objects o ON p.object_id = o.object_id
    INNER JOIN sys.partition_schemes ps ON ps.data_space_id = i.data_space_id
    INNER JOIN sys.partition_functions f ON f.function_id = ps.function_id
    INNER JOIN sys.destination_data_spaces dds ON dds.partition_scheme_id = ps.data_space_id AND dds.destinati
    INNER JOIN sys.filegroups fg ON dds.data_space_id = fg.data_space_id
    LEFT OUTER JOIN sys.partition_range_values rv ON f.function_id = rv.function_id AND p.partition_number = r
WHERE
    i.index_id < 2 AND o.object_id = OBJECT_ID('dbo.Log') AND rv.value <= @Max_id

print 'max_id: ' + cast(@Max_id as varchar)
print ''
print 'truncate command: ' + @truncate_sql
print ''
print 'merge command:' + @merge_sql

-- keep the execution commented until you have confirmed the partition range for truncation
--exec sp_executesql @truncate_sql
--exec sp_executesql @merge_sql
```

## Public Doc Reference

- Original blog post

### How good have you found this content?