http://doi.org/10.35784/iapgos.3058

received: 2.09.2022 | revised: 13.12.2022 | accepted: 15.12.2022 | available online: 30.12.2022

EFFICIENTLY PROCESSING DATA IN TABLE WITH BILLIONS OF RECORDS

Piotr Bednarczuk, Adam Borsuk
University of Economics and Innovation in Lublin, Institute of Computer Science, Lublin, Poland

Abstract: *Over time, systems connected to databases slow down. This is usually due to the increase in the amount of data stored in individual tables, counted even in the billions of records. Nevertheless, there are methods for making the speed of the system independent of the number of records in the database. One of these ways is table partitioning. When used correctly, the solution can ensure efficient operation of very large databases even after several years. However, not everything is predictable because of some undesirable phenomena become apparent only with a very large amount of data. The article presents a study of the execution time of the same queries with increasing number of records in a table. These studies reveal and present the timing and circumstances of the anomaly for a certain number of records.*

Keywords: systems aging, partitioning, efficiently data processing, billions of records

WYDAJNE PRZETWARZANIE DANYCH W TABELI Z MILIARDAMI REKORDÓW

Streszczenie: *Z biegiem czasu systemy podłączone do baz danych zwalniają. Wynika to zwykle ze wzrostu ilości danych przechowywanych w poszczególnych tabelach, liczonych nawet w miliardach rekordów. Niemniej jednak istnieją metody uniezależnienia szybkości systemu od liczby rekordów w bazie danych. Jednym z tych sposobów jest partycjonowanie tabel. Przy prawidłowym zastosowaniu rozwiązanie to może zapewnić wydajne przetwarzanie danych w bardzo dużych bazach danych nawet po kilku latach działania. Jednak nie wszystko jest tak przewidywalne ponieważ niektóre niepożądane zjawiska ujawniają się dopiero przy bardzo dużej ilości danych. W artykule przedstawiono badanie czasu wykonania tych samych zapytań przy rosnącej liczbie rekordów w tabeli. Badania te ujawniają i przedstawiają moment i okoliczności występowania anomalii dla pewnej liczby rekordów.*

Słowa kluczowe: starzenie się systemów, partycjonowanie, efektywne przetwarzanie danych, miliardy rekordów

Introduction

Billions of records in single tables are often found in data warehouses or BIG DATA databases. The administrators of such databases often face a optimalisation task of queries execution time. This is an important problem because it occurs in every large database. The ideal solution would allow, despite the increase of the number of records in the tables, to perform operations on the database as quickly as at the time of its implementation [2].

An example of such a solution that is implemented in most database engines is table partitioning [3].

The report presents a practical example of the use of partitioning on tables containing up to a billion records. The greatest advantage of partitioning was presented in the study which shows that the increase number of records in tables does not affect the query execution time.

The paper is an extension of publication [1] therefore appearing repetitions or similarities of the text in first four chapters result solely from the desire to present a coherent and complete the course of the research.

1. Table partitioning

Table partitioning is a division of tables into parts, physical files, constituting some separated ranges of data, e.g.: monthly, quarterly or annually. This division takes place in the database files, but the developer sees them as one object; one table. Although it is always possible to read the contents of each partition by using the appropriate tags, a standard queries e.g. SELECT, UPDATE and MERGE on the partitioned table will processing the data only from one or a few set of records, not all records.

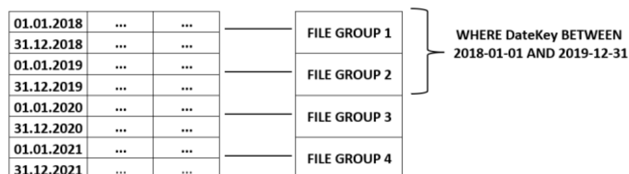

Fig. 1. Visualisation of table partitioning by yearly partitioning function

In this way, regardless of the increase in the number of records in the table, the number of records that the query will be performed on is almost always significantly smaller than the total size of the table. Then you will achieve the best performance and predictability of query execution times because it will usually relate only to a limited number of partitions, e.g. always 2 for data from the last 30 days. This is the main advantage and benefit of data partitioning.

2. Preparing for partitioning

The first step to start the implementation of partitioning in our database, is to first create a partitioning function. For this purpose, it is worth doing a short analysis of use cases to set the optimal period of partitioning. Based on my experience, there are three such most common periods: yearly, quarterly, and monthly, and in many of cases monthly is the best. This is why the monthly partitioning function was selected for the research, as can be seen in listing 1.

Listing 1. A monthly partitioning function

```
CREATE PARTITION FUNCTION PartitionFunctionByMonth (int)
AS RANGE RIGHT FOR VALUES(
             20150101
            ,20150201
            ,20150301
              :
              :
              :
            ,20201201
          );
```

Instead of date type, partitioning function is based on integer type and in this way monthly, quarterly or annual ranges are set. Using integer type can seems to be incorrect. This conscious optimization procedure used in data warehouses [5]. In queries of tables with billions of records, it matters what exactly data type the WHERE clauses will be based on. Queries will run much faster, with the integer type then date type. The next step is to create a partitioning scheme based on the partitioning function. The code responsible for creating the schema is shown in listing 2.

Listing 2. Partitioning scheme based on the monthly partitioning function

```
CREATE PARTITION SCHEME PartitionSchemaByMonth
AS PARTITION PartitionFunctionByMonth all to ([PRIMARY])
```

Having a function and a partitioning scheme the tables can be created where bilions of records will be generate. For this research there is created partitioned table was created FactResellerSalesPartitioned along with an identical table without

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Utwór dostępny jest na licencji Creative Commons Uznanie autorstwa – Na tych samych warunkach 4.0 Miedzynarodowe.

partitions FactResellerSales. Is worth to note that the integer type as a key has too small a range for a writing billion records, so the bigint type is selected.

To ensure the same measurement conditions of time executions for NSERT, SELECT, UPDATE, and MERGE queries, exactly the same data will be inserted into both tables. Measurements will be taken after added 100 million records added to both table each time. he structure of the partitioned table is presented in listing 3.

Listing 3. Partitioned table structure; the unpartitioned table will have the same structure

```
CREATE TABLE [dbo].[FactResellerSalesPartitioned](
            [SalesKey] [bigint] IDENTITY(1,1) NOT NULL,
            [DateKey] [int] NOT NULL,
            [EmployeeKey] [int] NOT NULL,
            [CustomerKey] [int] NOT NULL,
            [ProductKey] [int] NOT NULL,
            [SalesValue] dec(5,2) NOT NULL
          )
```

It is not without significance, is to use the partitioning scheme when creating a clustering index on a partitioned table (FactResellerSalesPartitioned).To perform this operation before importing data into the table because it may take a long time, especially on a table with a large amount of data. the code creating the index is presented in the listing 4.

Listing 4. Index using a partitioning scheme

```
CREATE UNIQUE CLUSTERED INDEX [PartitionedIndexReport] on
[dbo].[FactResellerSalesPartitioned]([SalesKey],[DateKey])
on PartitionSchemaByMonth ([DateKey])
```

Listing 5 shows the code responsible for creating an identical clustered index for a table without partitions (FactResellerSales). Except that we do not create this index in the partitioning scheme.

Listing 5. Definition of the Index on the table without partitioning

```
CREATE UNIQUE CLUSTERED INDEX [IndexReport]
on [dbo].[FactResellerSales]( [SalesKey],[DateKey])
```

Now we can consider that partitioning on (FactResellerSalesPartitioned) table has been enabled after created and executed function, schema and clustered index. You do not need to change of SELECT, UPDATE, and MERGE instruction to use partitioning. The only necessity is that the condition in the WHERE section is built on the DateKey column that was added to the partition key. The query optimizer reads data only from those partitions that are within the date range included in the WHERE clause [1, 4]. Listing 6 shows the use of partitioning in a SELECT query.

Listing 6. A SELECT query on a partitioned table is no different from a query on a table with no partitions

```
SELECT EmployeeKey, SUM(SalesValue)
FROM [dbo].[FactResellerSales] --unpartitioned table
WHERE DateKey between 20070925 and 20070927
GROUP BY EmployeeKey

SELECT EmployeeKey, SUM(SalesValue)
FROM [dbo].[FactResellerSalesPartitioned] -- partitioned table
WHERE DateKey between 20070925 and 20070927
GROUP BY EmployeeKey
```

As you can see in the example of SELECT query, it looks almost identical, only the name of the table changes. Therefore, only queries against partitioned tables will be presented for subsequent queries. Listing 7 shows the example use of partitions in the UPDATE query.

Listing 7. A UPDATE query on a partitioned table

```
UPDATE fs
SET SalesValue = t.SalesValue
FROM [dbo]. [FactResellerSalesPartitioned] fs
JOIN [dbo].[TMP] t on [fs].DateKey = [t].DateKey
AND [fs].[SalesKey] = [t].[SalesKey]
WHERE fs.DateKey between 20320421 and 20320520
```

The most complicated is use of partitioning on MERGE query – listing 8.

Listing 8. A MERGE query on a partitioned table

```
MERGE
    dbo.FactResellerSalesPartitioned AS [TargetTable]
            USING (
            SELECT
            [SalesKey]
          , [DateKey]
          , [EmployeeKey]
          , [CustomerKey]
          , [ProductKey]
          , [SalesValue]
            FROM [dbo].[TMP]
           ) [SourceTable]
          on [TargetTable].DateKey = [SourceTable].DateKey
          and [TargetTable].[SalesKey] = [SourceTable].[SalesKey]
          when matched and (
            [TargetTable].[SalesValue]!=[SourceTable].[SalesValue]
            )
          THEN UPDATE
          set [TargetTable].[SalesValue]=[SourceTable]. [SalesValue]
          WHEN NOT MATCHED BY TARGET
          THEN INSERT (
            [DateKey]
          , [EmployeeKey]
          , [CustomerKey]
          , [ProductKey]
          , [SalesValue]
          )
          VALUES (
            [SourceTable].[DateKey]
          , [SourceTable].[EmployeeKey]
          . [SourceTable].[CustomerKey]
          , [SourceTable].[ProductKey]
          , [SourceTable].[SalesValue]
    );
```

In the presented queries, the condition in the WHERE clause had to be based on the DateKey field. Because this is the field that was defined when creating the clustering index as the one after which partitioning will take place. This column must be of the same data type as that defined in the partitioning function. In this case it is an integer so dates are written as numbers, e.g., October 14, 2020 will be 20201014.

3. Method and conditions of research

The AdventureWorksDW 2017 database was used to carry out the measurements. This database is a training database that reflects data warehouses used in real systems. The test consisted in comparing the execution times of operations on a partitioned table and a table without a partition. Since the AdventureWorksDW 2017 database contained only tens of thousands of records, it was necessary to generate more data based on a combination of records from existing dimensions: product (DimProduct), sellers (DimEmploye) and customers (DimCustomer) and dates that already existed in the DimDate table. The generated data was then inserted in parallel into a partitioned (FactResellerSalesPartitioned) and non-partitioned (FactResellerSales) table.

One million records were inserted for each day in a random way that was a combination of data from the various dimensions:

- 1000 clients from DimCustomer with over 18,000 records,
- 10 sellers from DimEmployee dimension with 300 items,
- 100 products from the dimProduct dimension with over 600 items. Then the combinations of these data were inserted into the sales fact tables. This is how the daily data increase of one million records was created (1000 × 10 × 100 = 1,000,000).

The measurements were conducted beginning from 1 million records to 10 million and later every 100 million up to 1 billion records and every billion records to reach 10 billion. Like this for each significant number of records: 1, 10, 100, 200 ... 1000, 2000 … 10000 million, measurements were taken of the amount of time the SELECT, UPDATE, MERGE commands took to reading data from the last 30 days, i.e. 30 million records. A monthly partitioning function was used operations were performed on the last two partitions. The exception was when the table contained one million and 10 million records what means that was used only one partition.

4. Measurement of query execution times

The measurements concerned the execution times of the SELECT, UPDATE and MERGE queries from listing 6, 7 and 8 always for the last 30 million records added to the tables. The time was measured in simply way with a precision in milliseconds. The script of execution queries time calculated is shown in listing 9.

Listing 9. A script that measures the execution time of the SELECT, UPDATE and MERGE operation

```
DECLARE @start datetime
DECLARE @stop datetime
SELECT @start = getdate ();

    SELECT, UPDATE OR MERGE

@stop = getdate ()
SELECT DATEDIFF (ms, @start, @stop) /1000.00
```

It is worth noting that in this research, precision time measurement is not so important. It is mainly about checking whether the query execution time on a partitioned table will remain at the same level while the time of performing the same query on a table without a partition will increase. It does not matter if it is three seconds, five or ten seconds. It is only important that the query execution time on a partitioned table is independent of the data increment.

5. Results

The measurements times for the following SQL statements: INSERT, SELECT, UPDATE, MERGE are compare for a non-partitioned and partitioned table for configuration:

- hardware: Samsung 870 QVO drive, Intel Core i5-3210M processor, 2.5GHz, memory 8GB RAM,
- software: Microsoft SQL Server 2018 Standard Edition with recommended by Microsoft training database AdventureWorksDW 2017.

Always, for each SQL statements: SELECT, UPDATE, MERGE the same constant conditions were keeping for the table with and without partitions:

- the same table structures,
- identical structure of the cluster index, which included the fields SalesKey and DateKey,
- always the same number of partitions: 2,
- always the same number of records participating in the each operations – 30 millions – see chapter 3. Method and conditions of research
- exactly the same data in both tables.

The measurement results are presented in table 1.

For better readability, the comparing the execution times of INSERT, SELECT, UPDATE, MERGE queries are presented on separate graphs on a logarithmic scale (figure 2–5).

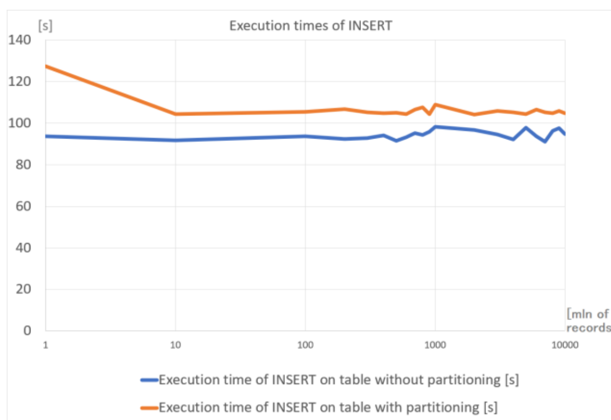

Fig. 2. Execution times of INSERT query on a table with and without partitioning, on a logarithmic scale

The execution time of the INSERT operation of the same number of records into a table with an increasing number of records is constant. For partitioning, it is only important that the INSERT execution time is also constant, and due to the use of the partitioning scheme and the partitioning function it is slightly longer.

Table 1. Execution times of INSERT, SELECT, UPDATE, MERGE query performed on tables with and without partitioning

Number of records	Execution time of							
	INSERT		SELECT		UPDATE		MERGE	
	on table without partitioning	on table with partitioning	on table without partitioning	on table with partitioning	on table without partitioning	on table with partitioning	on table without partitioning	on table with partitioning
[mln]	[s]	[s]	[s]	[s]	[s]	[s]	[s]	[s]
1	93.63	127.4	0.11	0.11	2.08	4.9	6.3	6.34
10	91.74	104.3	0.97	1	16.56	49.27	58.99	65.48
100	93.69	105.4	6.92	4.61	56.34	159.3	212.84	231.6
200	92.32	106.7	15.62	4.7	66.7	162.5	218.59	240.5
300	92.91	105.4	23.17	5.35	74.71	156.3	235.89	252.7
400	94.08	104.7	34.09	5.87	88.98	156.7	258.48	263
500	91.54	104.9	41.2	3.97	91.81	165.3	267.73	256.2
600	93.25	104.3	49.25	4.72	167.8	160.3	289.45	270.2
700	95.33	106.6	54.7	5.64	171.8	155.8	309.27	292.6
800	94.35	107.6	69.67	4.42	180.7	159.4	327.99	291.5
900	95.91	104.5	71.14	3.86	192.2	155.2	352.87	303.3
1000	98.22	108.9	81.57	5.54	197.6	159.9	374.99	337.8
2000	96.78	104.2	208.48	5.72	135.3	160.6	584.78	444.9
3000	94.56	105.8	307.11	5.45	137.3	154.5	776.89	547.8
4000	92.26	105.3	461.23	5.42	138.3	158.8	977.45	212.3
5000	97.88	104.4	513.26	5.73	138.8	156.3	1186.7	224.9
6000	93.78	106.5	644.78	5.12	139.2	156.8	1365.4	218.3
7000	91.11	105.4	724.59	5.33	143.8	158.9	1600.8	227.6
8000	96.27	104.7	852.17	5.53	140.8	163.6	1855	238.4
9000	97.54	105.9	1106.2	5.49	227.9	219.2	2062.7	297.5
10000	94.75	104.9	1160.7	5.88	289.2	348.3	2339.7	349.1

Fig. 3. Execution times of SELECT query on a table with and without partitioning, on a logarithmic scale

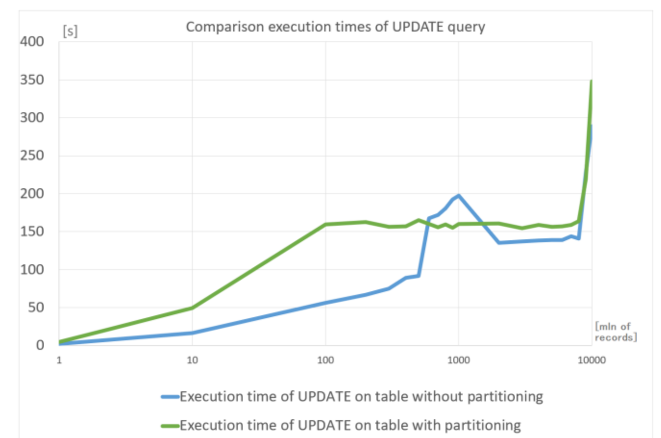

Fig. 4. Execution times of UPDATE query on a table with and without partitioning, on a logarithmic scale

As you can see in the figure 3, SELECT execution times for a partitioned table remain constant with exponentially increasing execution time of the same query on a table without partitioning.

UPDATE query times look similar for both partitioned and non-partitioned tables. Similarly, they also rise at the limit of 10 billion records. There is also a peak around three billions of records for a query on a table without partitioning (figure 4).

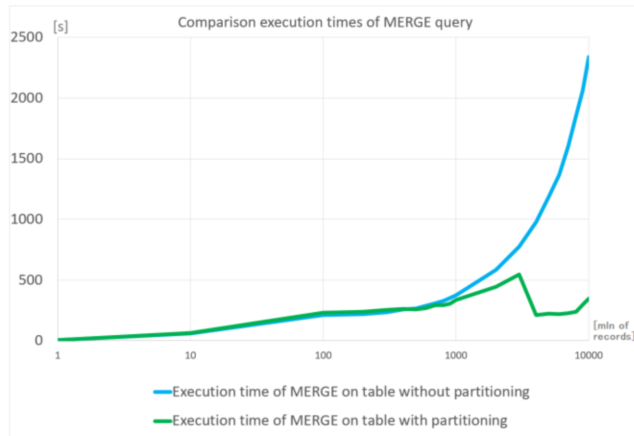

Fig. 5. Execution times of MERGE query on a table with and without partitioning, on a logarithmic scale

The results presented in table 1 and figure 3 clearly indicate that the SELECT query times from the partitioned table are relatively constant and fluctuate around 5-6 s. It is different for the table without partitioning, where this time always increases with the increase of data. The similar results is for the MERGE query where times from the partitioned table grows slightly to 350 s while the execution time on a non-partitioned table grows exponentially to 2340 s. I remind you that the tested SELECT query always operated on the same, invariable number of 30 million records.

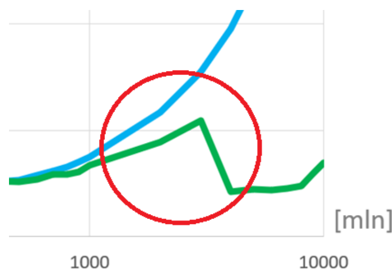

Fig. 6. The strange behavior of the MERGE query around three billions of records

That confirm that we do not lose much when arguing with tables without partitions on inserting and updating records, while we gain a lot from SELECT and MERGE queries on tables with partitions.

Noteworthy is the strange behavior of the MERGE query appearing from one billion of records, where a significant increase in processing time is visible, in the peak to over 500 s for three billions of records (figure 6). This may be due to the native optimization algorithm of the database engine or to hardware characteristic. This is what we will focus on in future research.

6. Conclusions

The research clearly indicate the partitioning table gives effectiveness of data processing in databases with billions of records. This is confirmed by the results of measurements of INSERT, SELECT, UPDATE, MERGE. We do not lose much with inserting and updating records, and we gain a lot from selecting and merging data on tables with partitions even on database with billions of records. The proposed solution solves the problem of "systems aging" with time when more and more records are added to the database. Thanks to partitioning, we can achieve the same system efficiency at the beginning, right after starting and after a few years of its implementation.

Additionally, when processing a large number, billions of records, an anomaly was noticed that to some extent deteriorates the SELECT and MERGE times on partitioned tables, it requires further investigation if the problems occur with a different hardware and software configuration.

References

[1] Bednarczuk P.: Optimization in very large databases by partitioning tables, Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska 10(3), 2020, 95–98.
[2] Bandle M., Giceva J., Neumann T.: To Partition, or Not to Partition, That is the Join Question in a Real System. International Conference on Management of Data, 2021.
[3] Kumar A., Jitendra Singh Y.: A Review on Partitioning Techniques in Database. International Journal of Computer Science and Mobile Computing 13(5), 2014, 342–347.
[4] Microsoft documentation, Data partitioning guidance, https://learn.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning
[5] Qi W., Song J., Yu-bin B.: Near-uniform Range Partition Approach for Increased Partitioning in Large Database. 2nd IEEE International Conference on Information Management and Engineering, 2010, 101–106.
[6] Song J., Bao Y.: NPA: Increased Partitioning Approach for Massive Data in Real-time Data Warehouse. 2nd International Conference on Information Technology Convergence and Services, 2010, 1–6.
[7] Tanvi J., Shivani S.: Refreshing Datawarehouse in Near Real-Time. International Journal of Computer Applications 46(18), 2012, 24–29.
[8] Zheng K. et al.: Data storage optimization strategy in distributed column-oriented database by considering spatial adjacency. Cluster Computing 20, 2017.

Ph.D. Eng. Piotr Bednarczuk
e-mail: Piotr.Bednarczuk@wsei.lublin.pl

He is a doctor in the Institute of Computer Science at the University of Economics and Innovation in Lublin. Studied and defended his Ph.D. thesis at Lublin University of Technology. He supports his scientific knowledge with professional practice gained in a leading IT company, where he has been working for over 15 years, currently as the head of the database solutions department in the mobile systems department. His research area focuses on the software engineering web database systems, mobile-device systems and databases and data warehouses.

http://orcid.org/0000-0003-1933-7183

M.Sc. Adam Borsuk
e-mail: adam.max.borsuk@gmail.com

He is a master degree absolvent of the Institute of Computer Science at the University of Economics and Innovation in Lublin. Studied programming and data analysis and defended his master's thesis in 2021. He supports his scientific knowledge with professional practice gained in a leading IT company, where he has been working for over 4 years, currently as the database developer.

http://orcid.org/0000-0003-2316-1694