"1st International Symposium on Computing in Informatics and Mathematics (ISCIM 2011)"
in Collabaration between EPOKA University and "Aleksandër Moisiu" University of Durrës
on June 2-4 2011, Tirana-Durres, ALBANIA

PERFORMANCE IMPROVEMENT THROUGH TABLE PARTITIONING
(Comparison of table partitioning in SQL Server 2008)

Julian FEJZAJ[1], Endri XHINA[1], Denis SAATCIU[1], Bora BIMBARI[1]

[1]*Department of Informatics, Faculty of Natural Sciences, University of Tirana, Albania*

Increasing the size of the databases might face database administrators with performance issues.
Most of the software vendors for DBMS products have included tools and techniques that help the database administrator to improve the performance of the database. In this article we will test one of the techniques used to enhance the database performance, named "table partitioning". The test will be done on SQL Server, which is one of the most used database management systems. The article will show the steps to implement the table partitioning in SQL Server 2008 R2. A partitioned table with two partitions will be created to test the performance of queries on each partition. A data population process will be applied to the table in order to fill the partitions with differentiated amount of data. The largest partition will be called "archive" and the smaller one "current".
A comparison table storing the amount of time required to execute each of the queries will be created. Six tests for each query will be executed in order to provide accurate results.
The comparison table will guide the interpretation process and will facilitate the conclusions.

Keywords: Database, Performance, Partitioning, SQL Server

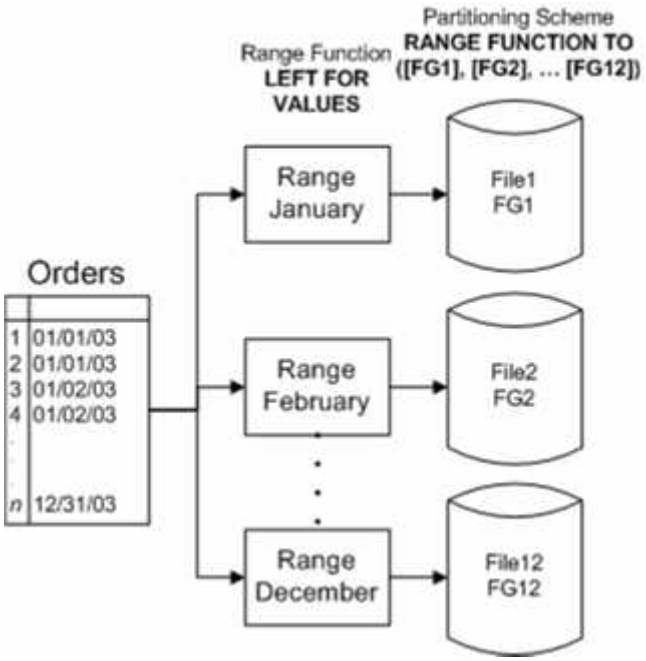
INTRODUCTION

In some applications the databases become huge and the tables store hundreds of GB of information. Table management is not a trivial task. That is why there are several techniques that aim to make the tables' management easier and improve the performance.
Table Partitioning is a technique that restructures the data stored in a table. A big table is separated into units called partitions. Each partition is stored separately from the other ones. Partitions of a table may be stored in the same file group as well as in different ones. Furthermore, the file groups might be stored in separate disks allowing in this way, parallel reading and writing of the data in different partitions.
Table partitioning helps in:

- managing big tables and their indexes easily
- improving performance by filtering queries and applying them on specific partitions and not on the whole table

What is table partitioning?

Partitioning allows you to partition the rows of a table based on a logical expression on a column(s). The column is called the partitioning key. Each partition can be stored on a separate file group. A file group is an object that has no storage on it. The file group groups one or more files where the actual rows of the table are stored. The files might be on separated disks allowing the rows of the table to be stored on separate disks depending on the value of the partitioning key.



A partitioned table might be created from the start; moreover, it is possible to partition an existing non partitioned table as well as modify the partitions of the exiting table , through splitting or merging existing partitions.

Picture no 1. shows schematically how a table storing the Orders is separated physically in three different partitions based on the value of order date.

As it is evident from the picture, there are three different files (File1, File2, File3) that support the partitioning.

Three file groups are created, FG1, FG2 and FG3, each containing a file, respectively File1, File2, File3.

Partitioning is based on the partitioning function, which defines: the data type of the partitioning key and the range of value. The partitioning function, once created might be reused.

The partitioning function does not refer to any physical location, nor does it reference any table or column name. In the example shown in the picture, the range function defines three ranges January, February, and March.
A partition scheme is created after the partitioning function. The partition scheme maps the ranges with the file groups. The partition scheme does not refer to any table or column that allows the partition scheme to be used by several tables.
Finally, when creating a table at the ON clause at the end of the create table function, we define that the table should be created on a partition scheme instead on the file group.

Note:
As a default the table is created at the Primary file group.

TOOLS AND METHODOLOGY

Hardware

The tests for this article will be executed on a Server with the specifications below:
HP DL360 G5 Server
Model Number: DL360 G5
CPU Speed: 3 GHz
Memory Size: 2 GB
Number of CPUs: 1
Operating Systems: Windows Server 2003 32-Bit

Database Server Software and Tools

The sample database is implemented in SQL Server 2008 R2 Enterprise Edition.
The query editor of SQL Server 2008 Management Studio is used as a query editor.
SQL Server Profiler is used to trace the queries and to display execution time for each query.

Methodology

A partitioned table with a key of type INT is created in two partitions.
The table is populated in order to have a current (small) and an archive (big) partition.
The same query has been executed six times and the average running time has been calculated, in order to have a more accurate evaluation.
Before executing a query SQL Server, buffers are cleaned up by deleting its cache.
Results are displayed in tables and correspondent charts are created.
Conclusions are based on this data.

EXPERIMENT

Below we will show the steps used in the experiment.

Creating the partition function

We create a partitioning function. The function defines how many partitions will be created and what will be the values for each of them.

```
CREATE PARTITION FUNCTION PartRange (INT)
AS RANGE LEFT FOR VALUES (1,2)
```

We call the partitioning function partRange. The key of the partitioning will be a column of data type int.
'Range left or right' defines whether values should go with the partition at the left or at the right.
We created 3 ranage.
Partition 1 – values <= 1
Partition 2 – values = 1,2
Partition 3 – values > 2

Creating the partition scheme

```
CREATE PARTITION SCHEME PartScheme AS
PARTITION PartRange
ALL TO ([PRIMARY])
```

The partitioning scheme (PartScheme) refers to the partitioning function PartRange. For the sake of this example, the partitions will be saved in the primary file group. The table can now be created.

Creating the partitioned table

```
CREATE TABLE PartTable
    (
    i INT ,
    s CHAR(8000) ,
    PartCol INT
    )
ON
 PartScheme (PartCol)
```

The table has a column PartCol of type INT. The value of this column will define in which partition a row of the table will be stored.
In order to make evident potential performance problems, the table has no Primary Key Index and the table contains a column of type char (8000) that would make a row of the table to occupy one page memory. Char is a fixed length data type, therefore columns would need 8000 bytes even if we store there a single character.

Note: As a default in an SQL Server memory page 8096 bytes can be stored there.
Populating the table

Let's insert some rows on the table

```
INSERT PartTable (i, s, PartCol) SELECT 1, 'a', 1
INSERT PartTable (i, s, PartCol) SELECT 2, 'a', 2
INSERT PartTable (i, s, PartCol) SELECT 3, 'a', 2
INSERT PartTable (i, s, PartCol) SELECT 4, 'a', 2
INSERT PartTable (i, s, PartCol) SELECT 5, 'a', 2
INSERT PartTable (i, s, PartCol) SELECT 6, 'a', 2
INSERT PartTable (i, s, PartCol) SELECT 7, 'a', 2
```

We enter some more rows at the table using the following loop:

```
DECLARE @i INT
SELECT @i = 13
WHILE @i > 0
BEGIN
    SELECT @i = @i - 1
INSERT PartTable (i, s, PartCol)
    SELECT i, s, PartCol
    FROM PartTable
    WHERE PartCol = 2
END
```

We select the data of each partition in order to verify that the rows are inserted into the right partitions.

```
SELECT *
FROM sys.partitions
WHERE OBJECT_ID = OBJECT_ID('PartTable')
```

The output of this query for the partitions that we are studying is shown below:

partition_id	object_id	index_id	partition_number	hobt_id	rows
72057594041532416	2117582582	0	1	72057594041532416	1
72057594041597952	2117582582	0	2	72057594041597952	16384

As we can see from the result, we notice that there are 16384 rows on partition 2 and 1 rows in partition1.

In order for us to compare the same partitioning scheme but with different amount of data, we create four different physical tables which are based on the same partitioning scheme – PartScheme, but with a different cardinality of rows. The

table below shows the number of rows for the two partitions for each of the tables. We are naming partition 1 current and partition 2 archive:

Table Name	Partition	Nr_rows
partTable	Archive	16385
	Current	4
partTable2	Archive	114688
	Current	8192
partTable3	Archive	344064
	Current	16384
partTable4	Archive	3440640
	Current	163840

Test Queries

On each table on each partition we executed two types of queries:

1- select * from <*table*> WHERE PartCol = x and i=<*unique value*>
2- select * from <*table*> WHERE PartCol = x

The first query retrieves on row from the partition.
The second query retrieves all rows from the partition.

These queries are run six times. After each execution the following SQL statements are executed, in order to clean the cache:

```
DBCC FREESYSTEMCACHE
DBCC FREESESSIONCACHE
DBCC FREEPROCCACHE
```

The execution times are stored in an excel table and the average is taken as the final execution time for each query. This execution time is used in the tables of the results section below. The query duration time is calculated in milliseconds.

RESULTS AND DISCUSSION

Many operations can apply at the partition level rather than the table level.
One comparison would be to execute a query that selects a row respectively on the archive and current partition. Please note that the queries below are executed against the same table.
The table and chart below shows the results of the queries

select * from PartTable WHERE PartCol = 2 and i=500
select * from PartTable WHERE PartCol = 1 and i=1

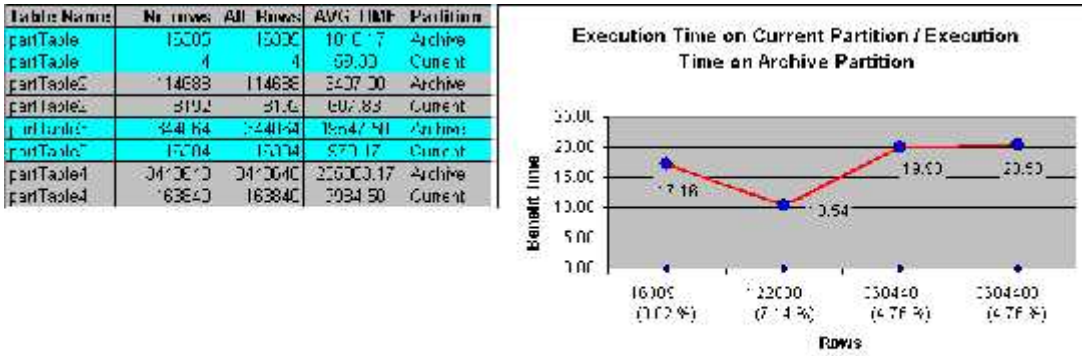


Chart no.1

As can be verified from chart no. 1, the benefit time with the size of the table growing has a tendency to grow linearly with the size of the table.

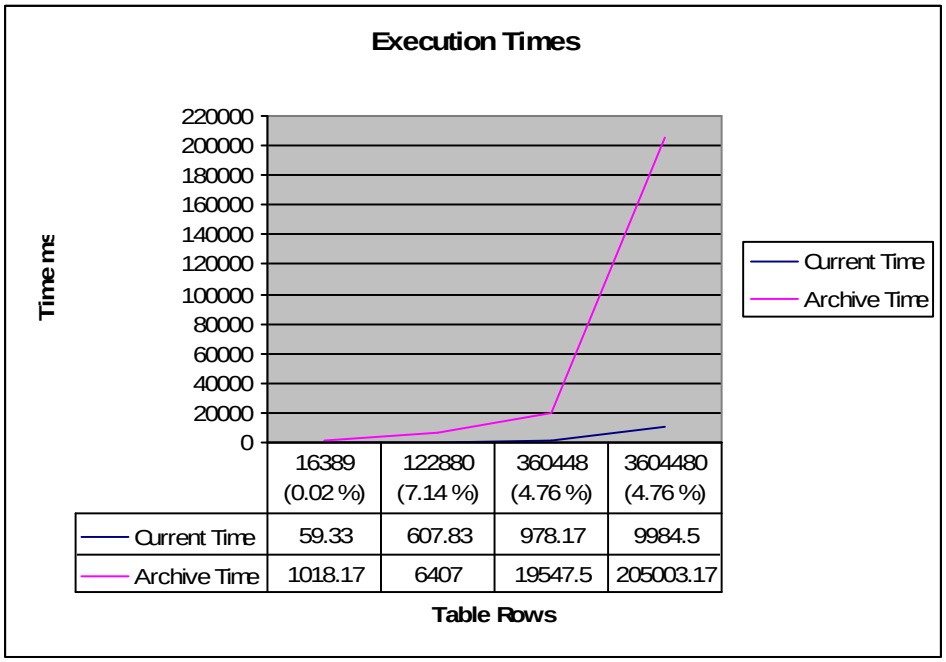


Chart no. 2

Chart no. 2 shows the execution time of the query that selects one row from the table. The query is executed against the same table in the current and archive partitions growing gradually the size of the partitions. As can be seen from the chart, the execution time with the growing of each partition grows as well as the

difference between execution time between the current and archive partition. This proves the performance benefit when partitioning a big table.

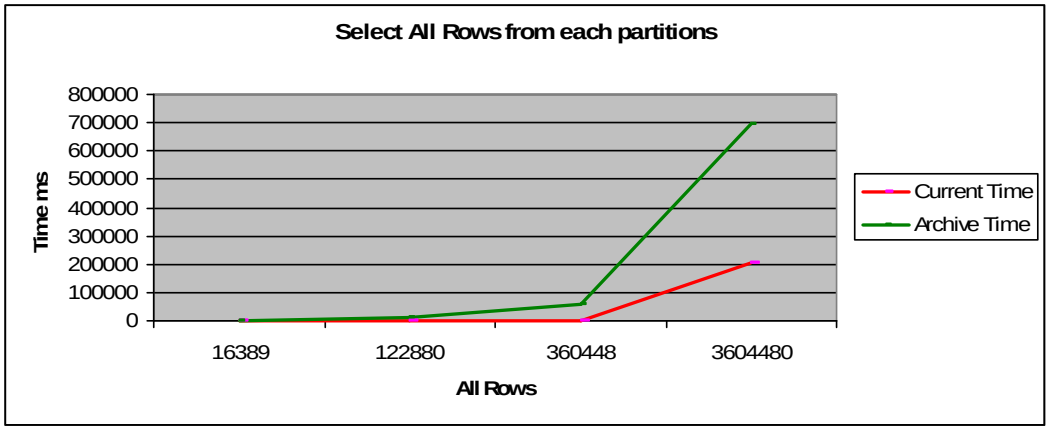


Chart no. 3

Chart no.3 shows the execution time of the query that selects all rows from the partition as presented in Table 1. The query is executed against the same table in the current and archive partitions growing gradually the size of the partitions. As the chart shows, the execution time with the growing of each partition grows as well as the difference between execution time between the current and archive partition. This proves the performance benefit when partitioning a big table.

select * from PartTable WHERE PartCol = 1
select * from PartTable WHERE PartCol = 2

Current Rows	Archive Rows	Total Rows	Current part %	Current Time	Archive Time	Archive/Report
4	16385	16389	0.024412572	91	1057	11.61538462
8192	114688	122880	7.142857143	1074	10759	10.01769088
16384	344064	360448	4.761904762	2291	56512	24.66695766
163840	3440640	3604480	4.761904762	201697	694753	3.444538094

Table 1

CONCLUSIONS

Table Partitioning is a technique that improves considerably the performance of access at big tables.

One advantage of this technique is the fact that the table rows are logically within the same table at the same time and the rows are stored in separate disks. Thus, this allows increase in performance and does not require any changes in the applications based on the database.

REFERENCES

[1] SQL Server 2008 Query Performance Tuning Distilled , Grant Fritchey, Sajal Dan, Apress,2009

[2] Building a Data Warehouse with Examples in SQL Server, Vincent Rairardi, Apress, 2008

[3] Microsoft SQL Server 2008 New Features , Michael Otey, McGrawHills, 2008

[4] Microsoft SQL Server 2008 Relational Databases Design and Implementation, Louis Davidson, APress, 2008