Query Optimization Techniques in Microsoft SQL Server

Costel Gabriel CORLĂŢAN, Marius Mihai LAZĂR,
Valentina LUCA, Octavian Teodor PETRICICĂ
University of Economic Studies, Bucharest, Romania
gabi.corlatan@yahoo.com, lazar_mariusmihai@yahoo.com,
luca.valentina@ymail.com, octavian.petricica@ymail.com

Microsoft SQL Server is a relational database management system, having MS-SQL and Transact-SQL as primary structured programming languages. They rely on relational algebra which is mainly used for data insertion, modifying, deletion and retrieval, as well as for data access controlling. The problem with getting the expected results is handled by the management system which has the purpose of finding the best execution plan, this process being called optimization. The most frequently used queries are those of data retrieval through SELECT command. We have to take into consideration that not only the select queries need optimization, but also other objects, such as: index, view or statistics.

Keywords: *SQL Server, Query, Index, View, Statistics, Optimization.*

1 Introduction

We consider the following problems as being responsible for the low performance of a Microsoft SQL Server system. After optimizing the hardware, the operating system and then the SQL server settings, the main factors which affect the speed of execution are:

1. Missing indexes;
2. Inexact statistics;
3. Badly written queries;
4. Deadlocks;
5. T-SQL operations which do not rely on a single set of results (cursors);
6. Excessive fragmentation of indexes;
7. Frequent recompilation of queries.

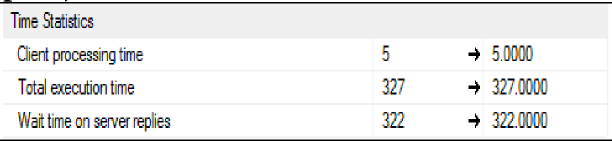
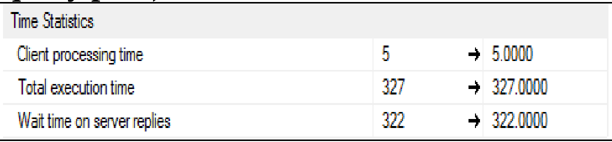
These are only a few of the factors which can negatively influence the performance of a database. Further, we will discuss each of the above situations and give more details.

2. Missing indexes

This particular factor affects the most SQL Server's performance. When missing indexing of a table, the system has to go step by step through the entire table in order to find the searched value. This leads to overloading RAM memory and CPU, thus considerably increasing the time execution of a query. More than that, deadlocks can be created when for example, session number 1 is running, and session number 2 queries the same table as the first session.

Let's consider a table with 10 000 lines and 4 columns, among which a column named ID is automatically incremented one by one.

Table 1.1. Running a simple query to retrieve a row in a table

With clustered index (execution time / query plan)	Without clustered index (execution time / query plan)
	

Time Statistics			
Client processing time	5	→	5.0000
Total execution time	327	→	327.0000
Wait time on server replies	322	→	322.0000

Time Statistics			
Client processing time	5	→	5.0000
Total execution time	327	→	327.0000
Wait time on server replies	322	→	322.0000

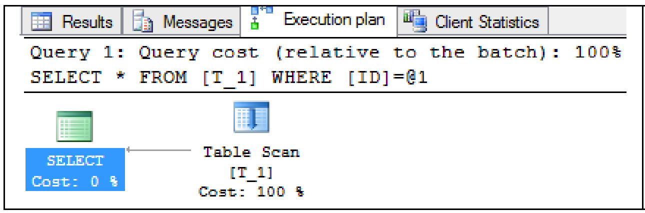
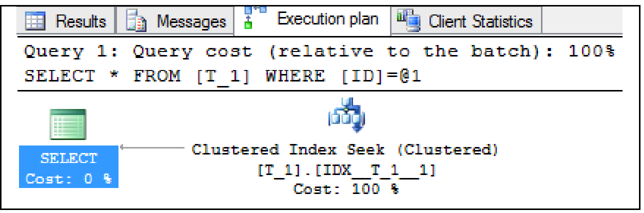
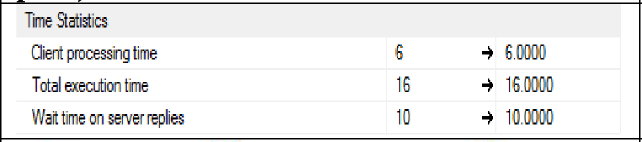
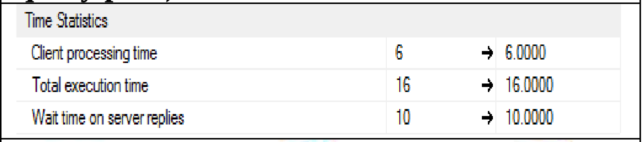
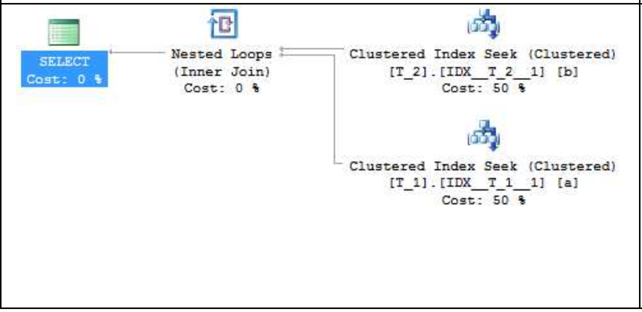
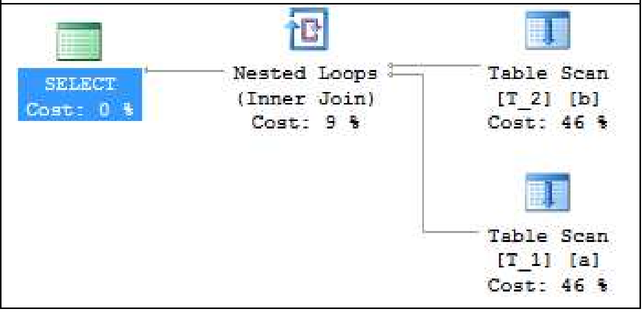
 Results | Messages | Execution plan | Client Statistics
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [T_1] WHERE [ID]=@1
SELECT Cost: 0 % ← Table Scan [T_1] Cost: 100 %
	 Results | Messages | Execution plan | Client Statistics
	Query 1: Query cost (relative to the batch): 100%
	SELECT * FROM [T_1] WHERE [ID]=@1
	SELECT Cost: 0 % ← Clustered Index Seek (Clustered) [T_1].[IDX__T_1__1] Cost: 100 %

Table 1.2. Running a 2 table join query

With clustered index (execution time / query plan)	**Without clustered index (execution time / query plan)**
 Time Statistics Client processing time 6 → 6.0000 Total execution time 16 → 16.0000 Wait time on server replies 10 → 10.0000	 Time Statistics Client processing time 6 → 6.0000 Total execution time 16 → 16.0000 Wait time on server replies 10 → 10.0000
 SELECT Cost: 0 % ← Nested Loops (Inner Join) Cost: 0 % ← Clustered Index Seek (Clustered) [T_2].[IDX__T_2__1] [b] Cost: 50 % ← Clustered Index Seek (Clustered) [T_1].[IDX__T_1__1] [a] Cost: 50 %	 SELECT Cost: 0 % ← Nested Loops (Inner Join) Cost: 9 % ← Table Scan [T_2] [b] Cost: 46 % ← Table Scan [T_1] [a] Cost: 46 %

Tabel 1.3. Running a junction between two tables

Query (Q1)	**Query (Q2)**
select*fromT_1whereID= 50000	select* fromT_1asa innerjoinT_2asb ona.ID=b.ID wherea.ID= 50000

In Table 1.1, the query is created using a single table, with and without a clustered index on the column specified in the WHERE clause (Q1). In the second table (Table 1.2), the query has two tables, a join on ID column of the two tables and a WHERE clause (Q2).

According to [1] and [3], SQL Server supports the following types of indexes:

- Clustered index;
- Nonclustered index;
- Unique index;
- Columnstore index;
- Index with included columns;
- Index on computed columns;
- Filtered index;
- Spatial index;
- XML index;
- Full-text index.

According to [2], the main index optimization methods are the following:

- It is recommended that created indexes to be used by the query optimizer. In general, grouped indexes are better used for interval selections and ordered queries. Grouped indexes are also more suitable for dense keys (more duplicated values). Because the lines are not physically sorted, queries which run using these values which are not unique, will find them with a minimum of I/O operations. Ungrouped indexes are more suitable for unique selections and for searching individual lines;
- It is recommended for ungrouped indexes to be created with as low density as possible. Selectivity of an index can be estimated using the selectivity formula: number of unique keys/ number of lines. Ungrouped indexes with selectivity less than 0, 1 are not efficient and the optimizer will

refuse to use it. Ungrouped indexes are best used when searching for a single line. Obviously, the duplicate keys force the system to use more resources to find one particular line;

- Apart from increasing the selectivity of indexes, you should order the key columns of an index with more columns, by selectivity: place the columns with higher selectivity first. As the system goes through the index tree to find a value for a given key, using the more selective key columns means that it will need less I/O operations to get to the leaves level of the index, which results in a much faster query;
- When an index is created, transactions and key operations in database are taken into consideration. Indexes are built so that the optimizer can use them for the most important transactions;
- It is recommended that we take into consideration at the time of index creation, that they have to serve the most often combining conditions. For example, if you often combine two tables after a set of columns (join), you can build an index that will accelerate the combination;
- Give up the indexes which are not used. If, following the analysis of the execution plans of queries which should use indexes we see they cannot actually be used, they should be deleted;
- It is recommended creating indexes on references to external keys. External keys require an index with unique key for the referred table, but we have no restrictions on the table that makes the reference. Creation of an index in the dependent table can accelerate checking the integrity of external keys which result from the modifications to the referred table and can improve the performance of combining the two tables;
- In order to deserve the rare queries and reports of users, we recommend creating temporary indexes. For example, a report which is ran only once a year or once a semester does not require a permanent index. Create the index right before running the reports and give it up afterwards, if that makes things happen faster than running the report without any indexes;
- For unblocking page for an index, a system procedure can be used: *sys.sp_indexoptions*. This forces the server to use blocking at line level and table level. As long as the line blockings do not turn too often into table blockings, this solution improves the performance in the case of multiple simultaneous users;
- Thanks to using multiple indexes on a single table by the optimizer, multiple indexes with a single key can lead to a better overall performance than an index with a compound key. That is because the optimizer can query the indexes separately and can combine them to return a set of results. This is more flexible than using an index with compound key because the index keys on a single column can be specified in any combination, which cannot be done in the case of compound keys. Columns which have compound keys have to be used in order, from left to right;
- We recommend using Index Tuning Wizard application, which will suggest the optimized indexes for your queries. This is a very complex tool that can scan tracking files collected by SQL Server Profiler in order to recommend the indexes that will improve the performance.

3. Inexact Statistics

According to [3], the SQL Server database management system relies mostly on cost based optimization, thus the exact statistics are very important for an efficient use of indexes. Without these, the system cannot estimate exactly the number of rows, affected by the query. The quantity of data which will be extracted from one or more tables (in the case of join) is important when deciding the optimization method of the query execution. Query optimization is less efficient when date statistics are not correctly updated.

The SQL Server query optimizer is based on cost, meaning that it decides the best data access mechanism, by type of query, while applying a selectivity identification strategy. Each statistic has an index attached, but there can be manually created statistics, on columns that do not belong to any index. Using statistics, the optimizer can make pretty reasonable estimates regarding the needed time for the system to return a set of results.

Indexed column statistics

The utility of an index is entirely dependent on the indexed column statistics. Without any statistics, the SQL Server cost-based query optimizer cannot decide which is the most efficient way of using an index. In order to satisfy this requirement, it automatically creates statistics on a index key every time the index is created. The required mechanism of data extraction in order to keep the cost low can use changing data. For example, if a table has a single row that matches some value which is unique, then using a nonclustered index makes sense. But if data changes, when adding a big number of rows with the same column value (duplicates), using the same index does not make any sense.

According to [5], SQL Server utilizes an efficient algorithm to decide when to execute the system procedure that updates the statistics, based on factors such as number of updates and table size:

- When inserting a line into an empty table;
- When inserting more than 1000 lines in a table that already has 1000 rows.

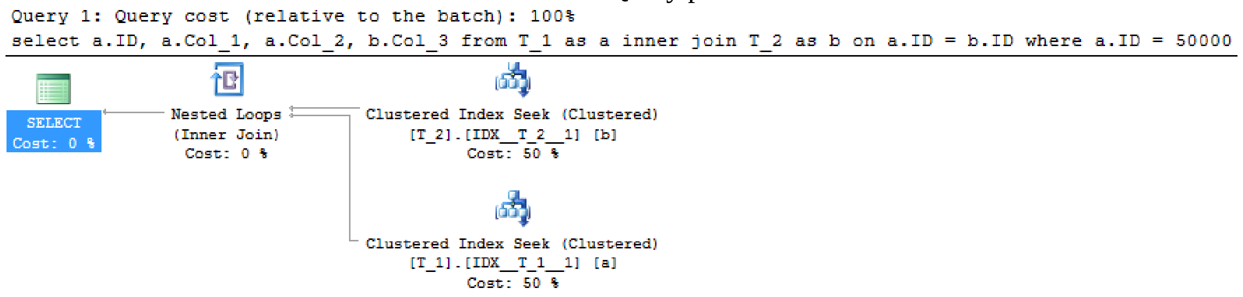
Automatic update of statistics is recommended in the vast majority of cases, except for very large table, where statistics updates can lead to slowing down or blocking the system. This is an isolated case and the best decision must be taken regarding its update.

Statistics update is made using the system procedure *sys.sp_updatestats* on an indexed table or view.

Unindexed column statistics

Sometimes there is the possibility of executing a query on an unindexed column. Even in this situation the query optimizer can take the best decision if it knows the data distribution of those columns. As opposed to index statistics, SQL Server can create statistics regarding the unindexed columns. Information regarding data distribution or the probability of having some value in an unindexed column can help the optimizer establish an optimum strategy. SQL Server benefits of query optimization even when it cannot use an index to locate the values. This automatically creates statistics on the unindexed column when the information that the system has, helps creating the best plan, usually when the columns are used in a predicate(ex: WHERE).

Table 1.4. Query plan



```
Query 1: Query cost (relative to the batch): 100%
select a.ID, a.Col_1, a.Col_2, b.Col_3 from T_1 as a inner join T_2 as b on a.ID = b.ID where a.ID = 50000
```

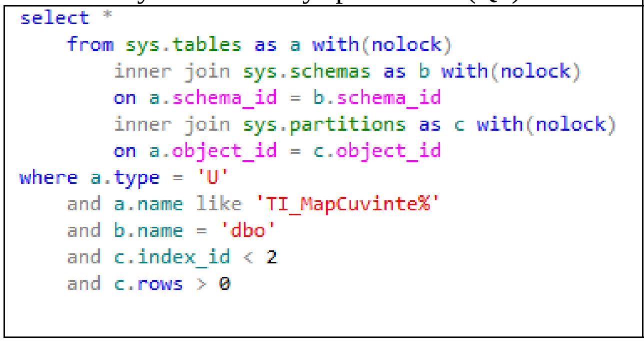
Table 1.5. Statistical data of the table "T_1"

```
selecta.ID,a.Col_1,a.Col_2,b.Col_3
       fromT_1asa
              innerjoinT_2asb
              ona.ID=b.ID
wherea.ID= 50000
```

4. Badly written queries

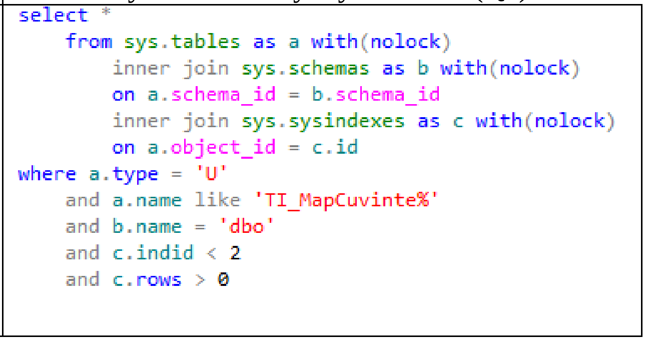
Index efficiency depends a lot on the way the queries are written. Taking a very large number of lines from a table can lead to inefficiency of the index. For improving performances, SQL queries must be written so that they use the existing indexes.

Table 1.6. T-SQL query to identify the names of the tables that contain at least one line using the system view "sys.partitions" (Q1)



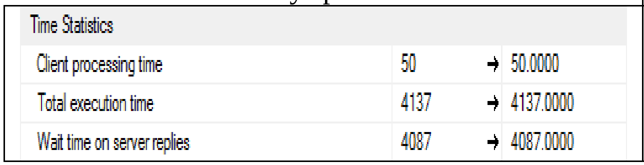
```
select *
    from sys.tables as a with(nolock)
        inner join sys.schemas as b with(nolock)
        on a.schema_id = b.schema_id
        inner join sys.partitions as c with(nolock)
        on a.object_id = c.object_id
where a.type = 'U'
    and a.name like 'TI_MapCuvinte%'
    and b.name = 'dbo'
    and c.index_id < 2
    and c.rows > 0
```

Table 1.7. T-SQL query to identify the names of the tables that contain at least one line using the system view "sys.sysindexes" (Q2)



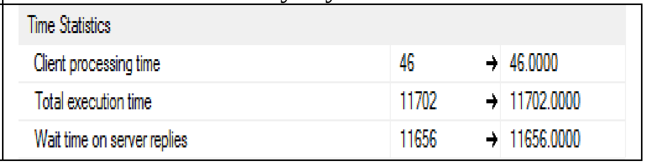
```
select *
    from sys.tables as a with(nolock)
        inner join sys.schemas as b with(nolock)
        on a.schema_id = b.schema_id
        inner join sys.sysindexes as c with(nolock)
        on a.object_id = c.id
where a.type = 'U'
    and a.name like 'TI_MapCuvinte%'
    and b.name = 'dbo'
    and c.indid < 2
    and c.rows > 0
```

Table 1.8. Query execution time using the system view "sys.partitions"



Time Statistics			
Client processing time	50	→	50.0000
Total execution time	4137	→	4137.0000
Wait time on server replies	4087	→	4087.0000

Table 1.9. Query execution time using the system view "sys.sysindexes"



Time Statistics			
Client processing time	46	→	46.0000
Total execution time	11702	→	11702.0000
Wait time on server replies	11656	→	11656.0000

As an example we select two T-SQL inquiries executed on system tables (views). Both return the names of the table which start with “TI Word Map” from the “dbo” layout and which contain at least one line. This method is more efficient than rolling a slider key on all tables from “sys.tables”, than rolling an inquiry “select count(*) from table name”, for each line with slider key, insertion of tables which contain at least one line in a temporary table, and then its inquiry. However this number can be determined by two methods: Table 1.6 and Table 1. 7. Although they may be identical the only difference between the two inquiries is that for determining the number of lines in Q1 table it extracts the number from sys.partitions and for Q2 from sys. sysindexes. As we may see Q1 is aproximatively three times quicker than Q2. In this case it is recommended using the system view sys.partitions rather than sys. sysindexes which according to Microsoft will be erased in future versions of SQL Server data bases.

Methods for optimizing SELECT option:

- Every time possible it is recommended to use as search columns in inquiries, the far left ones of the index. One index on col_1 and col_2 is of no help in an inquiry which filtrates results of col_2;
- It is recommended to build up WHERE terms which inquiry optimizer should recognize and use as searching tools;
- Don’t use DISTINCT or ORDER BY without any need. They may be used only to eliminate duplicate values or to

select a specific order in the result set. With the sole exception when the optimizer can find an index that might serve them, they can engage an intermediate working table, which can be expensive when talking about performance;

- Use UNION ALL instead of UNION when eliminating duplicates from a result set is not a priority. Because it eliminates the duplicates, UNION must sort or deal the result set before returning it;
- You may use SET LOCK_TIMEOUT when controlling the time limit a connection is waiting for a blocked resource. At the start of the session the automatic variable @@LOCK_TIMEOUT returns -1 which means that no value was selected to expire. You can select as a value for LOCK_TIMEOUT any positive number which establishes the number of milliseconds which an inquiry waits a blocked resource before to expire. In more difficult stages this is necessary to prevent apparent blocked applications;
- If an inquiry includes the IN predicate which contains a list of constant values (instead of a minor inquiry) order the values according to the release frequency in the exterior inquiry, more over when you know data tendencies. A common solution is alphabetical or numerical ordering of values but these may not be optimal. Because the predicate returns TRUE as soon as it reaches a resemblance for any of its values, moving on the first positions of the list the values which are released frequently should accelerate the inquiry, especially in the case when the column where the searching is done, is not indexed;
- It is recommended choosing algorithms despite of imbricated minor inquiries. A minor inquiry may need an imbricated inquiry that is a cycle in a cycle. In the case of imbricated error, the lines of the interior table are scanned for each line of the exterior table. This thing works very good for little tables and it was the only algorithm strategy used in SQL Server before 7.0 version, but as tables become bigger and bigger this solution becomes less and less efficient. It is much better to do the normal algorithms between tables and to let the optimizer select the best way to analyze them. Mostly the optimizer will try to transform the pointless minor inquiries in algorithms;
- When possible it is recommended to avoid CROSS JOINT type algorithms. With the exception of the case in which one cannot avoid the need for Cartesian product of two tables, it is used a more efficient method of algorithms to chain one table after the other. Returning an unwanted Cartesian product and then eliminating the duplicates generated by it using DISTINCT and GROUP BY is a problem which causes serious damage to the inquiry;
- You may use TOP(n) extension to restrict the number of lines returned by an inquiry. This thing is useful mainly when you may insert values using SELECT, because you may view only the values from the first part of the table;
- You may use OPTIONS clause of the SELECT instruction for influencing the inquiry optimizer with inquiry suggestions. As well you may select suggestions for tables and specific algorithms. As a rule the optimizer should optimize the inquiries, but there are cases in which the performing plan chosen is not the best. Using suggestions for inquiry, table or algorithm, you may entail to a certain type of algorithm, group or union to use a certain index, so on and so forth. These are called query hints; Here are some of these:
    - o FAST number_of_lines – points out that the inquiry is optimized

to rapidly reclaim the first "number of lines" After the first "number of lines" are returned, the inquiry goes on and produces the complete result set;
    - FORCE ORDER – points out that the syntax order of the inquiry is enabled during the inquiry optimization;
    - MAXDOP processor_number – overwrites the maximum number of configurated parallelism using sp_configure;
    - OPTIMIZE FOR (@variable_name) – points out to the inquiry optimizer to use a specific value to a certain local variable when inquiry is compiled and optimized;
    - USE PLAN – impels the inquiry optimizer to use an existent inquiry plan. It can be used with: insert, update, merge and delete options.
- Beside query hints there are also table hints, which influence the inquiry optimizer in taking some decisions as: using a blocking method for a table, using a certain index, blocking lines, etc. Here are some types:
    - NOLOCK – READUNCOMITTED and NOLOCK hints are applied only when data is blocked. They obtain Sch-s (stability scheme) blocked when compelling and executing. This indicates no blocking and doesn't stop other transactions accession to data, including their modification;
    - INDEX – impels using an index;
    - READPAST – points out to the system not to read the lines which are blocked by other transactions;
    - ROWLOCK – a line is blocked;
    - TABLOCK – points out that the blocking is at a table level;
    - HOLDLOCK – it's the equivalent of the SERIALIZABLE isolation level;
    - TABLOCKX – points out an exclusive blocking of the table.
- Testing and comparing two inquiries to determine the most efficient way of accessing data, one must be sure that the mechanism of data placement in cache memory of the SQL Server doesn't impair test results. One way of doing this is by cycling the server between inquiries rolling. Another way is by using undocumented DBCC commands to clean important cache memories. DBCCFREEPROCACHE extricates the memory for the procedure. DBCC DROPCLEANBUFFERS cleans all cache memories.

5. Excessive blocking (deadlocks)

According to [2], SQL Server is a software compliant with "ACID" rules where "ACID" is an acronym for atomicity, consistency, isolation and durability. This assures that the variations made by the simultaneous transactions are worthy isolated one versus the other. Compliance with the rules by the transactions is mandatory for data safety.

- ***Atomicity*****:** one transaction is atomic if compliant with the principle "all or nothing". When a transaction is successful all its variations become permanent, when a transaction fails, all the variations are canceled;
- ***Consistency*****:** one transaction is atomic if compliant with the principle "all or nothing". When a transaction is successful all its variations become permanent, when a transaction fails, all the variations are canceled;
- ***Isolation*****:** a transaction is isolated if it doesn't concern other transactions or it isn't concerned by other rival

transactions on same data (levels of isolation);

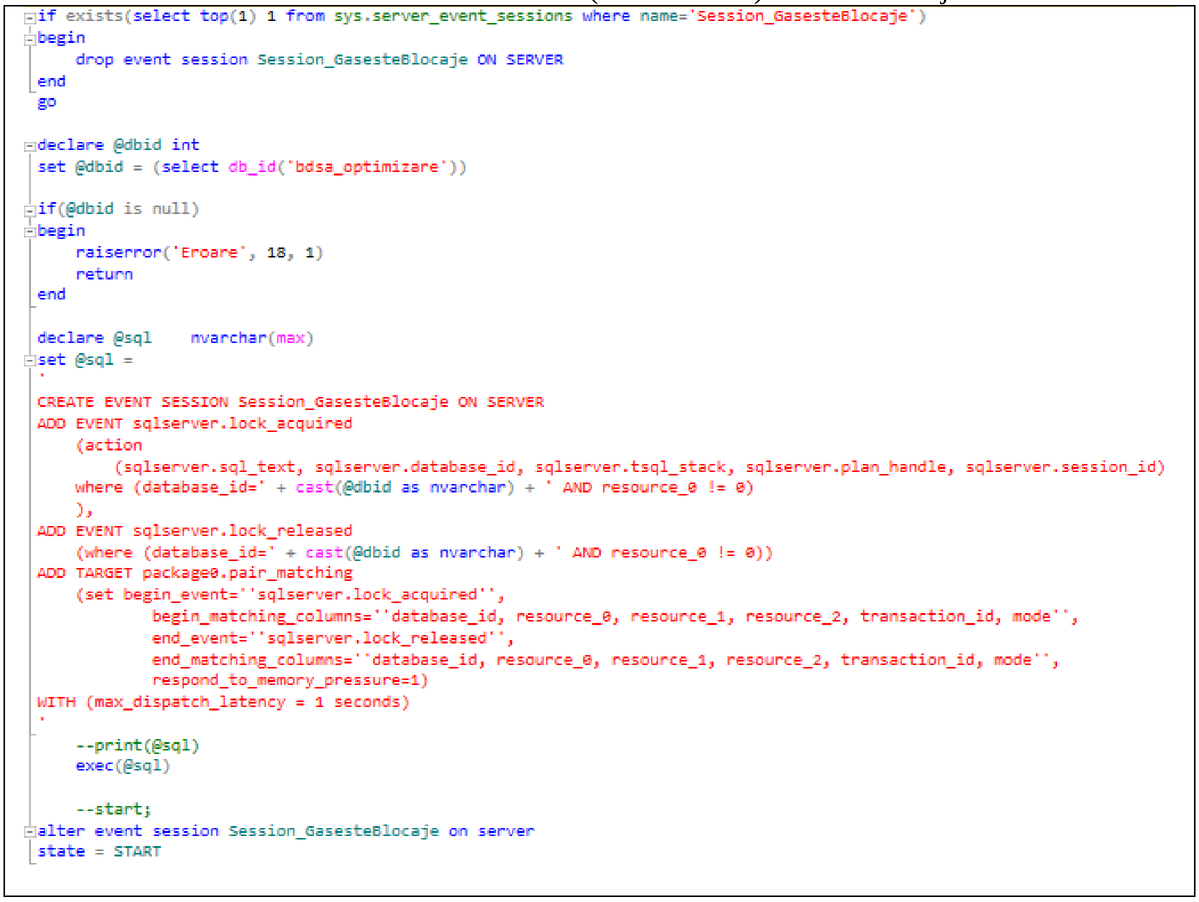
- ***Durability:*** a transaction is durable if it can be done or reversed despite of a system breakdown.

SQL Server Blocking

When a session is doing an inquiry, the system determines the resources of the data base and gives data base blocking for the respective session. The inquiry is blocked in case that a session got blocked. Despite all these, for offering isolation and rivalry SQL Server offers different levels of blocking as:

- *Row (RID*) – this blocking is done on a single row in a table and is the smallest blocking level. For example when an inquiry changes a row in a table a RID blocking is made for that inquiry.
- *Key (KEY)* – this is a blocking system of a row in an index and is identified as a blocking key. For example for a table with a clustered index, data pages of the table and data pages of the clustered index are the same. Since both rows are the same for the clustered index table only a KEY type blocking is obtained on the clustered index row or the limited set of rows while accessing the rows in the table;
- *Page (PAG)* – PAG blocking system is kept only on a page of a table or index. When an inquiry requires more rows in a page, the consistency of all required rows can be kept whether by RID or KEY blocking on a row level or by PAG blocking on a page level;
- *Extent (EXT*) – this type of blocking is made after using ALTER INDEX REBUILD command. This command is used at a table level and the table pages can be moved from one scale to another scale. All this time the integrity of the extent is protected by EXT blocking;
- *Heap or B-tree (HoBT*) – a HEAP or B-TREE blocking is used when data is heaped on many file groups. Target object may be a table without clustered index or a B-TREE object. A setting in ALTER TABLE offers a certain level of control over the blocking mode (lock escalation). Because the parts are heaped in many file groups, each has to have its own data assignation definition. HoBT blocking type acts on a partition level, not on a table level;
- *Table (TAB)* – this blocking type is the highest blocking level. Books complete access to the table and its indexes;
- *File (FIL)*;
- *Application (APP)*;
- *MetaData (MDT)*;
- *Allocation Unit (AU)*;
- *Database (DB)* – it is a blocking system kept on a database level. When an application gets access to a data base, the blocking manager offers a blocking at a data base level worthy to a SPID (Speed Process ID).

Table 1.10. The session (event session) for a database jams



```
if exists(select top(1) 1 from sys.server_event_sessions where name='Session_GasesteBlocaje')
begin
    drop event session Session_GasesteBlocaje ON SERVER
end
go

declare @dbid int
set @dbid = (select db_id('bdsa_optimizare'))

if(@dbid is null)
begin
    raiserror('Eroare', 18, 1)
    return
end

declare @sql    nvarchar(max)
set @sql =
'
CREATE EVENT SESSION Session_GasesteBlocaje ON SERVER
ADD EVENT sqlserver.lock_acquired
    (action
        (sqlserver.sql_text, sqlserver.database_id, sqlserver.tsql_stack, sqlserver.plan_handle, sqlserver.session_id)
    where (database_id=' + cast(@dbid as nvarchar) + ' AND resource_0 != 0)
    ),
ADD EVENT sqlserver.lock_released
    (where (database_id=' + cast(@dbid as nvarchar) + ' AND resource_0 != 0))
ADD TARGET package0.pair_matching
    (set begin_event=''sqlserver.lock_acquired'',
            begin_matching_columns=''database_id, resource_0, resource_1, resource_2, transaction_id, mode'',
            end_event=''sqlserver.lock_released'',
            end_matching_columns=''database_id, resource_0, resource_1, resource_2, transaction_id, mode'',
            respond_to_memory_pressure=1)
WITH (max_dispatch_latency = 1 seconds)
'
    --print(@sql)
    exec(@sql)

    --start;
alter event session Session_GasesteBlocaje on server
state = START
```

Isolation levels of transactions

SQL Server accepts four types of isolation of a transaction. As I mentioned earlier the level of isolation of a transaction controls the way in which it affects and is affected by other transactions. In a level of isolation it is always a reverse relation between data consistency and users rivalry. Selecting a more restrictive level of isolation amplifies data consistency to the detriment of accessibility. Selecting a less restrictive level of isolation amplifies rivalry to the detriment of data consistency. It is important to balance these opposite interests, so the needs of the application to be assured.

- *READ UNCOMMITTED* - READ UNCOMMITTED parameter specification is essentially the same as using NOLOCK option in each table referenced by a transaction. This is the least restrictive of the four isolation levels in SQL Server. It allows "dirty reads" (reading not completed changes of other transactions) and readings that cannot be repeated (data which changes between readings during a transaction);
- *READ COMMITTED* - The default isolation level in SQL Server, so if you do not specify otherwise, you get the READ COMMITTED level. READ COMMITTED level avoids reading "dirty", enforcing shared locks on accessed data but allows modification of basic data during a transaction, thus the possibility of repeated readings and / or phantom data;
- *REPEATABLE READ* - The REPEATABLE READ level generates locks that prohibit other users to modify data accessed by a transaction, but does not prohibit the insert of new rows, which may result in phantom rows between readings during a transaction;
- *SERIALIZABLE* - The level SERIALIZABLE prevents "dirty reads" and phantom lines by

introducing key-range locks on data access. It is the most restrictive of the four isolation levels in SQL Server. This is equivalent to using the option HOLDLOCK on each table referenced by a transaction.

In order to enforce an isolation level for a transaction, the command SET TRANSACTION ISOLATION LEVEL must be used. The valid parameters for the isolation levels are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE.

6. T -SQL operations that are not based on a single set of results (cursors)

Given that Transact -SQL is a set based language, we know that it operates on sets. Writing non-set based code can lead to excessive use of cursors and loops. To improve performance, it is recommended to use queries based on set, not the row by row approach, as the latter leads to hardware overloading, resulting in a much higher data access time. Excessive use of cursors increases the stress on SQL server and results in reduced system performance.

Classification cursors

Based on the level of isolation, according to [3] and [4] cursors may be:

- ***Read-only*** – the cursor cannot be updated;
- ***Optimistic***– updatable cursor and uses the optimistic concurrency model (does not lock rows of data);
- ***Scroll Locks*** – updatable cursor that has a locking system for any row to be updated.

Types of cursors

- ***Forward-only*** – the default cursor type. It returns the rows from the data set sequentially. No extra space needed in tempdb and changes on data are visible as soon as the cursor reaches it;
- ***Static* –** static cursors return a result set that can only be read, which is impervious to changes in data. They are sometimes referred to as insensitive and are the opposite of dynamic cursors;
- ***Dynamic*** – as with forward- only cursors, dynamic cursors type reflect changes on rows as they reach them. No extra space in tempdb is needed, and unlike forward-only cursors they are scrollable, which means they are not restricted to sequential access on rows. Sometimes these cursors are called sensitive;
- ***Keyset*** – keyset cursors return a fully scrollable result set, whose membership and order are fixed. As with static cursors, the unique-key values set of the cursor lines is copied to tempdb when opening the cursor. This is why the cursor membership is fixed.

Cost comparison

Read-Only cursors

The read-only model has the following advantages:

- The lowest level of locking: the read-only model introduces the lock with the least impact and database synchronization. Locking on the base row while the cursor loops the rows can be avoided by using the NO_LOCK command in the SELECT instruction, but the dirty reads must be taken into account;
- The highest level of concurrency: as supplementary locks are not held in the rows that form its base, the read-only cursor does not block other users from accessing the base table.

The main disadvantage of the read-only cursor refers to the lack of updates: contents of the base table cannot be modified by the cursor.

Optimistic cursors

The optimistic model has the following advantages:

- Low level of locking - Similar to the read-only model, the optimistic concurrency model does not have a specific type of lock. In order to further improve the concurrency, the NOLOCK locking instruction may also be used, as in the case of the read-only concurrency model. Using the cursor to update a row requires exclusive rights on that row;
- High concurrency - Since only a shared lock is used on the rows behind it, the cursor does not block other users from accessing the base table. Changing a baseline will block other users from accessing the row during the update.

Scroll Locks cursors

The scroll locks model has the following advantages:

- Concurrency control: By blocking the base row corresponding the last row of the cursor, it ensures that the base row cannot be changed by another user;
- For very large data sets, the use of asynchronous cursors is recommended. Returning a cursor allows further processing while the cursor is populated. Asynchronous cursors are set as follows:

```
EXECsp_configure'show
advanced options', 1;
GO
RECONFIGURE
GO
EXECsp_configure'cursor
threshold', 0;
GO
RECONFIGURE
GO
```

- When configuring read-only result sets, unidirectional, FAST_FORWARD cursor option is recommended instead of FORWARD_ONLY. FAST_FORWARD option creates a FORWARD_ONLY and a READ_ONLY cursor with several built in performance optimizations;
- It is recommended to avoid changing a large number of lines using a cursor cycle contained in a transaction because each line that it changes may remain locked until the end of the transaction, depending on the transaction isolation level.

7. Excessive index fragmentation

Normally, data is organized in an orderly way. However, in the case in which the pages contain fragmented data or contain a small amount of data due to frequent page division, the number of read operations needed to return the data will be higher than usual. Increasing the number of readings is due to fragmentation.

Fragmentation occurs when table data is changed .When executing instructions to insert or update data (INSERT or UPDATE statements), clustered indexes on tables and nonclustered indexes are affected. This issue can cause a tear in the index leaf page when an index change cannot be stored on the same page. A new leaf page will then be added as part of the original page in order to maintain the logical order of rows in the index key. Although the new leaf page maintains the logical order of rows of data from the original page, it usually will not be physically adjacent to the original page on disk. For example, suppose that an index has nine key values (or rows in the index) and average index rows size allows a maximum of four rows in a leaf page. 8KB leaf pages are connected to previous and next pages to maintain the logical order of the index In Figure 1.1, leaf pages layer 1 is shown. Since the key values of index leaf pages are always sorted, a new row of the index with a key value of 25 must occupy a place between the extended key values of 20 and 30. Since the leaf page containing that existing index is filled with four rows, the new index row will result in dividing the page.

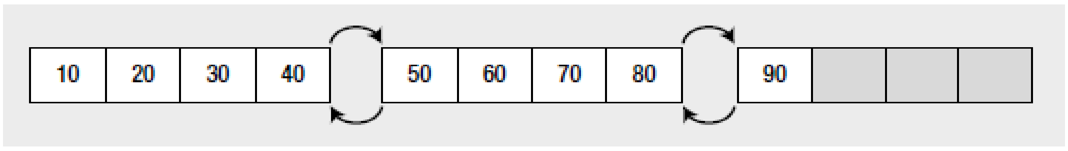


Fig 1.1. Layer leaf pages

A new page will be assigned to the index and a part of the first page will be moved on this new page so that the new index key can be expressed in the correct logical form. The links between indexed pages will also be updated as the pages are logically linked in index order. As it can be seen in Figure 1.1, a new page, even if linking it to the other pages id done in the correct logical order, physically the linking can sometimes be unordered.

Pages are grouped into larger units called extents, which may include eight pages. SQL Server uses unit as a physical unit of disk allocation. Ideally, the physical order of extensions contains leaf pages whose key must be the same as the logical order of the index. This reduces the number of required switches between extents when reading a series of rows in the index. However, crevice in the pages can disturb the pages within extensions physically and can also cause physical disorder even among extensions. For example, suppose the first two leaf pages of the index are as measure 1 and the third page is measure 2. If measure 2 contains unallocated space, then the new leaf page allocated to the index causes page division. The page will be measure 2, as shown in Figure 1.3.

With leaf pages distributed between two extents, one would ideally expect a read of a series of index rows with a maximum of one switch between the two extents. However, the disorganization of pages between extents can cause more than one switch during the read of a series of rows in the index. For example, to retrieve a number of rows in the index between 25 and 90, you will need three switches between the two extents, as follows:

- Firstly, a measure switch to retrieve the value of key 30 after the value of key 25;
- Secondly, a measure switch to retrieve the value of key 50 after the value of key 45;
- Thirdly, a measure switch to retrieve the value of key 90 after the value of key 80.

This type of fragmentation is called external fragmentation. Fragmentation can also occur in an index page. If an INSERT or UPDATE operation creates a page break, then free space will be left behind in the original leaf page. Free space can also be caused by a DELETE operation.

The net effect is the reduction of the number of rows included in one leaf page. For example, in Figure 1.3, the page split caused by the INSERT operation has created a gap in the first leaf page. This is known as internal fragmentation.

For a highly transactional database, it is preferable to deliberately leave space inside the leaf pages to be able to add new rows or change existing rows size without causing a rift in the page. In Figure 1.3 the free space of the first leaf page allows a key value in the index of 26. This can be added to the leaf page without causing a rift.

Heap pages can become fragmented in exactly the same way. Unfortunately, because of the storage mechanism and because every nonclustered index uses the physical location of the data to retrieve data from heap, defragmenting is quite difficult. The ALTER TABLE command using the REBUILD clause can be used to perform a reconstruction.

SQL Server 2012 exposes leaf pages and other data through a dynamic system view management system called sys.dm_db_index_physical_stats. It stores both index size and degree of fragmentation.

According to Microsoft, the commands ALTER INDEX REORGANIZE and ALTER INDEX REBUILD are used depending on the degree of fragmentation of the system view as follows:

Table 1.11. Degree of fragmentation of the system

Value of avg_fragmentation_in_percent column	**T-SQL code**
> 5% and <= 30%	ALTER INDEX REORGANIZE
> 30%	ALTER INDEX REBUILD

Rebuilding an index can be performed either online or offline, but reorganization can only be executed online.

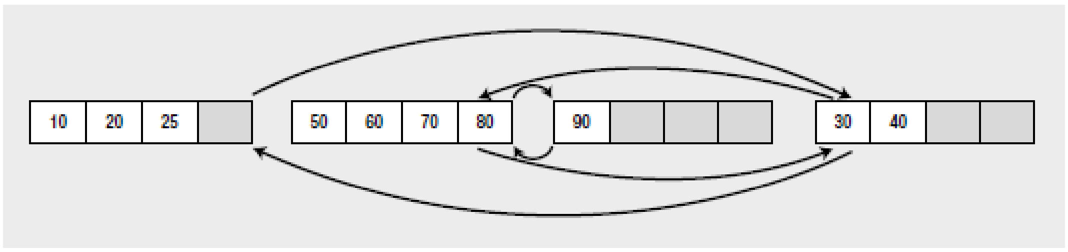


Fig 1.2. Outside leaf page order

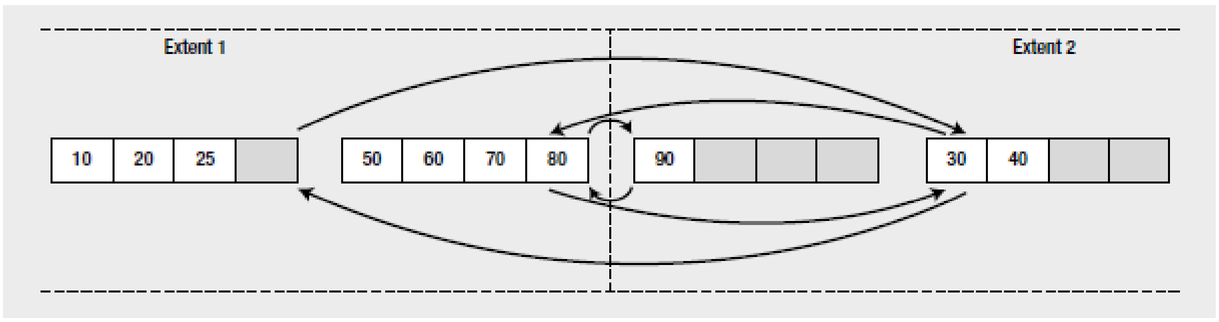
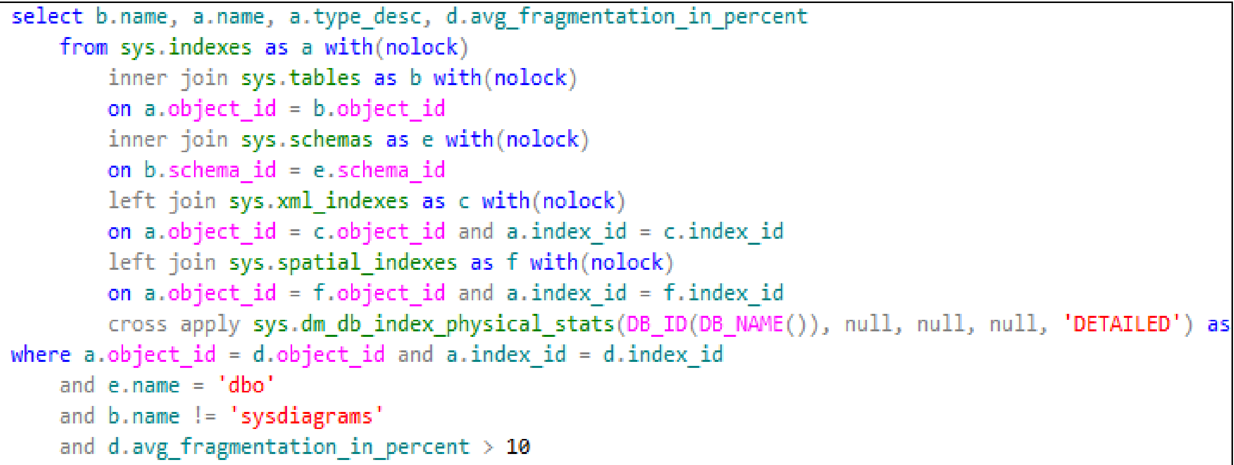


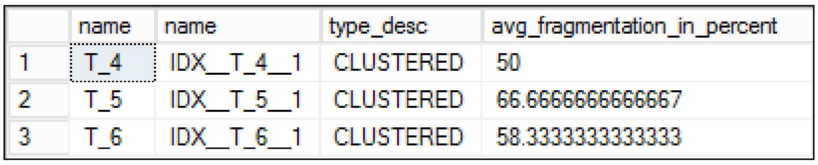
Fig 1.3. Outside leaf pages order distributed extensions

Note that this index fragmentation is different from disk fragmentation. Index fragmentation can't be determined by running the Disk Defragmenter tool, because the order pages in a SQL Server file is understood only by SQL Server, not by the operating system.



```
select b.name, a.name, a.type_desc, d.avg_fragmentation_in_percent
    from sys.indexes as a with(nolock)
        inner join sys.tables as b with(nolock)
        on a.object_id = b.object_id
        inner join sys.schemas as e with(nolock)
        on b.schema_id = e.schema_id
        left join sys.xml_indexes as c with(nolock)
        on a.object_id = c.object_id and a.index_id = c.index_id
        left join sys.spatial_indexes as f with(nolock)
        on a.object_id = f.object_id and a.index_id = f.index_id
        cross apply sys.dm_db_index_physical_stats(DB_ID(DB_NAME()), null, null, null, 'DETAILED') as
where a.object_id = d.object_id and a.index_id = d.index_id
    and e.name = 'dbo'
    and b.name != 'sysdiagrams'
    and d.avg_fragmentation_in_percent > 10
```

Fig 1.4. Determining the degree of fragmentation of the index tables in schema "dbo" with fragmentation greater than 10



	name	name	type_desc	avg_fragmentation_in_percent
1	T_4	IDX__T_4__1	CLUSTERED	50
2	T_5	IDX__T_5__1	CLUSTERED	66.6666666666667
3	T_6	IDX__T_6__1	CLUSTERED	58.3333333333333

Fig 1.5. The result of the query to determine the degree of fragmentation of indexes

8. Rebuilding Frequent Queries

The most common way to provide a reusable execution plan, independently of the variables used in a query, is to use a stored procedure or parameterized query. By creating a stored procedure to execute a set of SQL queries, the database system creates a parameterized execution plan independently of the parameters during execution. The execution plan generated will be reusable only if SQL Server does not have to recompile individual statements from stored procedure each time it is executed (e.g. sequences of dynamic SQL). Rebuilding frequent query execution causes time increases.

Optimizing stored procedures methods:

- Whenever possible it is recommended to use stored procedures instead of ad-hoc queries. In order to reuse the execution plan of an ad hoc SQL query you have to match exactly and must fully qualify each object meant. If in future use the query, everything is different: the parameters, name objects, key elements of SET, the plan will not be reused. A good solution that avoids the limitations of ad-hoc queries is to use the system stored procedure sys.sp_executesql. This is somewhere between rigid stored procedures and ad hoc Transact-SQL queries, allowing to run ad-hoc queries with replaced parameters. This facilitates reuse of ad-hoc execution plans without the need for precise consistency;
- For a small portion of a stored procedure the query plan must be rebuilt at every execution (e.g. due to data changes that doesn't make the optimal plan), but we do not want the overload associated with rebuilding plan for the entire procedure each time, that portion it should be moved in a stand-alone procedure. This allows reconstruction of its execution plan every time a run, but without affecting the procedure longer. If this is not possible, try using EXEC() to call the suspect code in the main procedure. Because this subroutine it is dynamically built we can generate a new execution plan at every execution, without affecting the whole stored procedure query plan;
- When possible, it is better to use output parameters of stored procedures instead of set results. If you need to return the

result of a calculation or to locate a single value in a table, it is preferable to return the output parameter of a stored procedure instead of a set result with a single line. Even if you return multiple columns, output parameters of stored procedures are more effective than complete set results;

- When you need to return a set of lines from a stored procedure to another, it is better to use output parameters of the cursor instead of set results. This technique is considerably more flexible and allows the second procedure to run more quickly since it doesn't work as set results. Then the caller can process the rows returned by the cursor as desired;
- It is recommended to minimize the number of network packets between the client and server. A very effective way to achieve this goal is to disable DONE_IN_PROC messages. You can disable it at the procedure level with the SET NOCOUNT command or at the server level with tracking indicator 3640. Doing so may lead to huge differences in performance, especially when relatively slow networks are used such as WAN networks. When you choose not to use the tracking indicator 3640, SET NOCOUNT ON should be used at the beginning of any stored procedures that you write;
- When adjusting query use PROCCACHE DBCC command to list information about cache memory reserved for the procedure. Also use the DBCC FREEPROCCACHE command to clear the memory cache so that multiple executions of a given procedure not alter test results. DBCC FLUSHPROCINDB used to force the creation of new execution plans for basic procedures.

Conclusions

Performance optimization is an ongoing process. This process requires continuous monitoring and improving database performance. The purpose of this paper is to provide a list of SQL scenarios to serve as a quick and easy reference guide during the development phase and maintenance of the database. Transact-SQL language provides a wide range of techniques for updating query. On the top of the list are found correct database design, adding indexes and query interrogations. Performance optimization is a complex subject, which certainly could fill many books. The secret to success is knowing the instruments mentioned above in relation to the available space, knowing how the server works and the own the required skills to solve problems using this knowledge.

References

[1] Adam Jorgensen, Jorge Segarra, Patrick Leblanc, Jose Chinchilla, Aaron Nelson, *Microsoft SQL Server 2012 Bible*, Ed. Johs Wiley & Sons, Inc., 2012, Indianapolis, Indiana - USA, ISBN: 978-1-118-10687-7

[2] Ken Henderson, *Transact-SQL (*Titlul original: *The Guru's Guide to Transact-SQL)*, Ed. Teora, București, România, 2002, ISBN: 973-20-0612-9

[3] Grant Fritchey, *SQL Server 2012 Query Performance Tuning,* Ed. Apress, USA, 2012, ISBN: 978-1-4302-4203-1

[4] Leonard Lobel, Andrew Brust, *Programming Microsoft SQL Server 2012*, Ed. Microsoft Press, 2012, ISBN-13: 978-0735658226

[5] Jason Strate, TedKrueger, *Expert Performance Indexing for SQL Server 2012*, Ed. Apress, USA, 2012, ISBN: 978-1-4302-3741-9

Costel Gabriel CORLĂŢAN studies at Academy of Economic Studies, Database for Business Support master program. Main technologies that he is working with are: MS SQL Server, .NET Framework, ASP. NET, C#.

Marius Mihai LAZĂR: Database for Business Support masterand and programmer in the area of database developing, tuning and optimization, mainly Microsoft technologies like SQL Server and Visual Studio. Also he develop web, desktop and mobile applications. In 2012 he published and documented an application about “Elementary cellular automaton”.

Valentina LUCA graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Bucharest Universisty of Economic Studies in 2012. Currently she is a masterand enrolled in Databases for Business Support program. Her fields of interest: databases, data warehouses, business intelligence.

Octavian Teodor PETRICICĂ: graduated from the Faculty of Mathematics and Informatics of the Hyperion University of Bucharest in 2012. He is currently a masterand of Database for Business Support program. His scientific fields of interest include: Databases, Web development and Application programming interfaces. At present he is a Software Developer in the department of IT Applications at Cargus International SA.