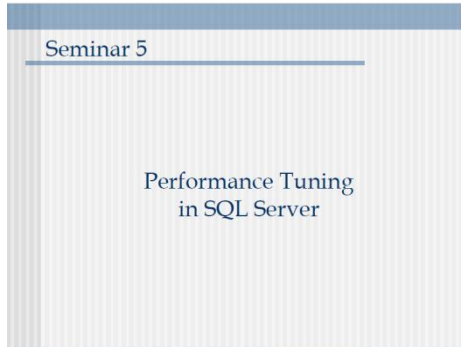


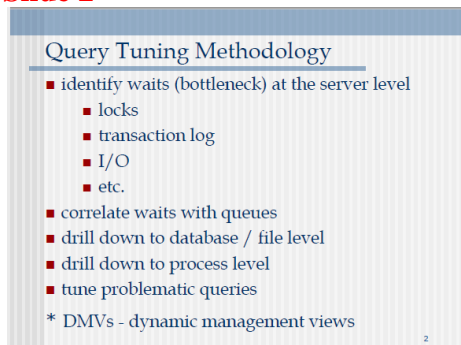
Seminar 5 – prezentare slide-uri

Slide 1



This seminar is going to introduce us to the Performance Tuning in SQL Server.

Slide 2



The main ideas that we will discuss here, refer to: the query tuning methodology, that includes the identification of the waits (or bottlenecks) at the server level, including locks, transaction log, input/output operations and so on, then, the correlation between the waits with queues, the drill down to database and file level, the drill down to process level, the tune problematic queries.

Dynamic Management Views, or, “DMVs” are query structures built into SQL Server that deliver details about server and database health/performance. DMVs provide a common mechanism to extract from SQL the performance data. There are multiple DMV that return configuration information and performance data, and some of those are remembered here. DMVs are powerful and flexible, but requires time.

Related to the Dynamic Management View Queries, we should take into account some elements:

1. Most queries return information quickly and with minimal overhead.
2. The query structures evolve with each SQL Server release with both new queries as well as changes to existing queries (i.e. new columns) added.
3. Queries return configuration detail (i.e. OS and SQL version). The information is not only helpful in terms of demographics, but is also extremely useful when used as part of conditional scripting.
4. Queries return Windows performance metrics as well detail about the inner workings of SQL Server. This data will show improper database tuning as well as poor server performance.

To optimize the performance on the database and also on the queries, there are some tips:

- Don't use the * in your queries. A SELECT * does an overload on the table, I/O and network bandwidth.
- All columns involved in indexes should appear on WHERE and JOIN clauses on the same sequence they appear on index.

- Use VIEWS only when are benefits of doing them.
- Verify if a critical query gains performance by turning it in a stored procedure.
- Avoid too much JOINS on your query: use only what is necessary!
- Avoid cursors at all costs!
- Always restrict the number of rows and columns of your result. It saves disk, memory and network of the database server. Always verify the WHERE clause and use TOP if necessary.
- Verify if the server is not suffering from not-enough-disk-space illness. Reserve at least 30% of available space on your disc.
- SQL Server is *case insensitive*: It does not care about A or a. Save time and don't use functions like LOWER and UPPER when comparing VARCHARs.
- The decreasing performance order of operators is: = (faster) >, >=, <, <=, LIKE, <> (slower)
- If a query is slow and the index is not being used by it (check in execution plan), it can be forced by using WITH (INDEX=index_name), right after the table declaration on the FROM clause.
- Use EXISTS or NOT EXISTS instead of IN or NOT IN. IN operators creates a overload on database.
- Try to use BETWEEN instead of IN, too.
- When using LIKE operator, try to leave the wildcards on the right side of the VARCHAR.
- Avoid to use functions on the queries. SUBSTRING is the enemy. Try to use LIKE instead.
- Queries with all operations on the WHERE clause connected by ANDs are processed from the left to right. So, if a operation returns false, all other operations in the right side of it are ignored, because they can't change the AND result anyway. It is better then to start your WHERE clause with the operations that returns false most of the time.
- Sometimes is better to make various queries with UNION ALL than a unique query with too much OR operations on WHERE clause.
- When there is a HAVING clause, it is better to filter most results on the WHERE clause and use HAVING only for what it is necessary.
- If there is a need of returning some data fast, even if it is not the whole result, use the FAST option.
- Use, if possible, UNION ALL instead of UNION. The second eliminates all redundant rows and requires more server's resources.
- Use less subqueries, or try to nest all of them on a unique block.
- Avoid to do much operations on your WHERE clause. If you are searching for $a + 2 > 7$, use $a > 5$ instead.
- Use more variable tables and less temporary tables.
- Use functions to reuse code. But don't exaggerate on using them!
- To delete all rows on a table, use TRUNCATE TABLE statement instead of DELETE.
- For IDENTITY on a primary key, when are done dozens of simultaneous insertions on, make it a non-clusterized primary key index to avoid bottlenecks.

The query performance tuning in SQL Server refers to optimize the query performance. This one implies: improve the indexes, create highly-selective indexes, create multiple-column indexes, avoid indexing small tables, use indexes with filter clauses, use the query optimizer, understand response time vs. total time, and others.

To be able to troubleshoot performance problems one needs to know how SQL Server works when the queries are executed:

- first, the application sends a request to the server, containing a stored procedure name or some T-SQL statements.
- second, the request is placed in a queue inside SQL Server memory.
- third, a free thread from SQL Server's own thread pool picks up the request, compiles it and executes it.
- forth, the request is executed statement by statement, sequentially. A statement in a request must finish before the next starts, always. Stored procedures executes the same way, statement by statement. Statements can read or modify data. All data is read from in-memory cache of the database (the buffer pool). If data is not in this cache, it must be read from disk into the cache. All updates are written into the database log and into the in-memory cache (into the buffer pool), according to the Write-Ahead Logging protocol. Data locking ensures correctness under concurrency. When all statements in the request have executed, the thread is free to pick up another request and execute it.

A request is either executing or is waiting for something (being suspended) - is the key to troubleshooting SQL Server performance. If requests sent to SQL Server take a long time to return results then they either take a long time *executing* or they take a long time *waiting*. Knowing whether is one case or the other is crucial to figure out the performance bottlenecks. Additionally, if the requests take a long time waiting, we can dig further and find out *what* are they waiting for and for how long.

Waiting requests have Wait Info data Whenever a request is suspended, for *any* reason, SQL Server will collect information about *why* it was suspended and for how long. In the internal SQL Server code there is simply no way to call the function that suspends a task without providing the required wait info data. And this data is then collected and made available for you in many ways. This wait info data is crucial in determining performance problems.

Slide 3

Identify Waits

- `sys.dm_os_wait_stats`
 - returned table
 - wait_type
 - resource waits (locks, latches, network, I/O), queue waits, external waits
 - waiting_tasks_count
 - wait_time_ms
 - max_wait_time_ms
 - signal_wait_time_ms
 - reset DMV values
 - `DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);`

First, we are going to discuss about the Waits, more exactly, how to identify the waits.

The first DMV that we present here is `sys.dm_os_wait_stats`. This one returns information about all the waits encountered by threads that executed. This aggregated view is used to diagnose performance issues with SQL Server and also with specific queries and batches. Also, shows the time for waits that have completed. This dynamic management view (DMV) does not show current waits but helps to understand wait stats. It returns a table with the following columns: `wait_type`, that has resource waits, like, locks, latches, network, I/O, queue waits and external waits; `waiting_task_count`; `wait_time_ms`; `max_wait_time_ms`; `signal_wait_time_ms`.

Column name	Data type	Description
<code>wait_type</code>	nvarchar(60)	Name of the wait type.
<code>waiting_tasks_count</code>	bigint	Number of waits on this wait type. This counter is

		incremented at the start of each wait.
wait_time_ms	bigint	Total wait time for this wait type in milliseconds. This time is inclusive of signal_wait_time_ms.
max_wait_time_ms	bigint	Maximum wait time on this wait type.
signal_wait_time_ms	bigint	Difference between the time that the waiting thread was signaled and when it started running.

The types of Waits used are:

Resource waits - occur when a worker requests access to a resource that is not available because the resource is being used by some other worker or is not yet available. Examples: locks, latches, network and disk I/O waits. Lock and latch waits are waits on synchronization objects.

Queue waits - occur when a worker is idle, waiting for work to be assigned. Are most typically seen with system background tasks (i.e. the deadlock monitor, deleted record cleanup tasks). These tasks will wait for work requests to be placed into a work queue. Queue waits may also periodically become active even if no new packets have been put on the queue.

External waits - occur when a SQL Server worker is waiting for an external event (i.e. an extended stored procedure call, a linked server query) to finish. In the diagnose blocking issues, that external waits do not always imply that the worker is idle, because the worker may actively be running some external code.

Not in the end, here, we need to know, that the DMV values can be reset, by using DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR).

Next, I am going to show you some examples relative to sys.dm_os_wait_stats

[1_sys.dm_os_wait_stats_ok](#)

Slide 4

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
 - object_name - the category of the counter
 - counter_name - the name of the counter
 - instance_name - the name of the specific instance of the counter; often contains the name of the database
 - cntr_value - the current value of the counter
 - cntr_type - the type of the counter (as defined by the Windows performance architecture)

Next, we will discuss about the correlation of the waits with the queues. The DMV that we use is called sys.dm_os_performance_counters. This one is a system Dynamic Management View (DMV) that returns one row for each SQL Server performance counter. It is useful for obtaining information about current performance counter values. These counter values are also shown in Windows Performance Monitor. The permission needed to query this view is VIEW SERVER STATE. The columns returned are: object_name, counter_name, instance_name, cntr_value and cntr_type.

<i>Column name</i>	<i>Data type</i>	<i>Description</i>
object_name	nchar(128)	Category to which this counter belongs.
counter_name	nchar(128)	Name of the counter.
instance_name	nchar(128)	Name of the specific instance of the counter. Often contains the database name.

cntr_value	bigint	Current value of the counter. For per-second counters, this value is cumulative. The rate value must be calculated by sampling the value at discrete time intervals. The difference between any two successive sample values is equal to the rate for the time interval used.
cntr_type	int	Type of counter as defined by the Windows performance architecture.

If the return value is 0 rows -> means that the performance counters have been disabled -> look at the setup log and search for error 3409, "Reinstall sqlctr.ini for this instance, and ensure that the instance login account has correct registry permissions." -> This denotes that performance counters were not enabled. The errors immediately before the 3409 error should indicate the root cause for the failure of performance counter enabling.

On SQL Server, requires VIEW SERVER STATE permission. On SQL Database, requires the VIEW DATABASE STATE permission in the database.

Slide 5

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- **> 500 counters:** Access Methods, User Settable, Buffer Manager, Broker Statistics, SQL Errors, Latches, Buffer Partition, SQL Statistics, Locks, Buffer Node, Plan Cache, Cursor Manager by Type, Memory Manager, General Statistics, Databases, Catalog Metadata, Broker Activation, Broker/DBM Transport, Transactions, Cursor Manager Total, Exec Statistics, Wait Statistics, etc.
- **cntr_type = 65792** → **cntr_value** contains the actual value

If, there are more than 500 counters, we talk about the Access Methods, User Settable, Buffer Manager, Broker Statistics, SQL Errors, Latches, Buffer Partition, SQL Statistics, Locks, Buffer Node, Plan Cache, Cursor Manager by Type, Memory Manager, General Statistics, Databases, Catalog Metadata, Broker Activation, Broker/DBM Transport, Transactions, Cursor Manager Total, Exec Statistics, Wait Statistics, and so on.

Some counters are repeated in different categories. In total there are 405 different counters. The information that is usually overlooked, but that requires attention is the **cntr_types** column. It defines the type of the counter and thus the method that should be used to calculate the current counter value.

The performance counters exposed by SQL Server are invaluable tools for monitoring various aspects of the instance health. The counter data is exposed as a shared memory object for the windows performance monitoring tools to query. The counter data exposed in the view are in a raw form (-> needs to be interpreted appropriately before it can be used). The **cntr_type** column value indicates how the values have to be interpreted. The type of each counter is indicated in the **cntr_type** column as a decimal value. The distinct values are:

Decimal	Hexadecimal	Counter type define
1073939712	0x40030500	PERF_LARGE_RAW_BASE
537003264	0x20020500	PERF_LARGE_RAW_FRACTION
1073874176	0x40020500	PERF_AVERAGE_BULK
272696576	0x10410500	PERF_COUNTER_BULK_COUNT
65792	0x00010100	PERF_COUNTER_LARGE_RAWCOUNT

Next, I will talk a little bit about each of these values.

For `cntr_type=65792`, the `cntr_value` returned contains the actual value.

The `cntr_types` column value for the `PERF_COUNTER_LARGE_RAWCOUNT` counter type is 65792 (0x00010100 – in hexadecimal). This counter value shows the last observed value directly, not the average value. It is usually used to monitor object counts (if one is monitoring a counter type 65792, the value got in the `counter_value` column when query the view, is the current value of the counter; no additional calculation is required).

Slide 6

Correlate Waits with Queues

- `sys.dm_os_performance_counters`
- `cntr_type = 537003264` → `cntr_value` contains real-time results, which are divided by a “base” to obtain the actual value; by themselves, they are useless
 - to get a ratio: divide by a “base” value
 - to get a percentage: multiply the result by 100.0

6

For `cntr_type=537003264`, the `cntr_value` contains real-time results, which are divided by a “base” to obtain the actual value; by themselves, they are useless. To obtain a ratio, one has to divide by a “base” value, and to obtain a percentage, one has to multiply the results by 100.0

The `cntr_types` column value for the `PERF_LARGE_RAW_FRACTION` counter type is 537003264 (0x20020500 – in hexadecimal). This counter value represents a fractional value as a ratio to its corresponding `PERF_LARGE_RAW_BASE` counter value. These counters show a ratio (presented in percents), (i.e. fraction between two values – the `PERF_LARGE_RAW_FRACTION` counter and its corresponding `PERF_LARGE_RAW_BASE` counter value). Additional calculation is needed to find out the value used for monitoring and troubleshooting performance issues.

Slide 7

Correlate Waits with Queues

- `sys.dm_os_performance_counters`
- `cntr_type = 272696576`
 - time-based, cumulative counters
 - a secondary table can be used to log intermediate values

7

For `cntr_type=272696576`, we have time-based, cumulative counters and a secondary table can be used to log intermediate values.

The `cntr_types` column value for the `PERF_COUNTER_BULK_COUNT` counter type is 272696576 (0x10410500 – in hexadecimal). This counter value represents a rate metric. The `cntr_value` is cumulative. The value is obtained by taking two samples of the `PERF_COUNTER_BULK_COUNT`

value. The difference between the sample values is divided by the number of seconds between the samples. This provides the value per second rate. It is important to know how long the sample period is, otherwise, one would not be able to calculate the value per second (usually a 5-minute period is used). The formula for the current metric value is $(A2-A1)/(T2-T1)$, where A1 and A2 are the values of the monitored PERF_COUNTER_BULK_COUNT counter taken at sample times T1 and T2, and T1 and T2 are the times when the sample values are taken.

Slide 8

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- `cntr_type = 1073874176` and `cntr_type = 1073939712`
- poll both the 1073874176 counter value and the base value (the 1073939712 counter)
- poll both values again (e.g., after 10 seconds) ☺
- to obtain the desired result, compute:

$$\text{UnitsPerSecond} = (cv2 - cv1) / (bv2 - bv1) / 10$$

For `cntr_type=1073874176` and `cntr_type=1073939712`, one has to poll both the 1073874176 counter value and the base value, poll both values again (after 10 seconds), to obtain the desired result, that is computed like: $\text{UnitsPerSecond} = (cv2-cv1)/(bv2-bv1)/10$.

The `cntr_types` column value for the PERF_LARGE_RAW_BASE counter type is 1073939712 (0x40030500 – in hexadecimal). This counter value is raw data that is used as the denominator of a counter that presents a instantaneous arithmetic fraction (or denominator for further calculation). These counters collect the last observed value. The counters of this type are only used to calculate other counters available via the view. All counters that belong to this counter type have the word base in their names, so it is not a counter that provides useful info, it's just a base value for further calculations.

The `cntr_types` column value for the PERF_AVERAGE_BULK counter type is 1073874176 (0x40020500 – in hexadecimal). This counter value represents an average metric. The **cntr_value** is cumulative. The base value of type PERF_LARGE_RAW_BASE is used which is also cumulative. The value is obtained by first taking two samples of both the PERF_AVERAGE_BULK value `cv1` and `cv2` as well as the PERF_LARGE_RAW_BASE value `bv1` and `bv2`. The difference between `cv1` and `cv2` and `bv1` and `bv2` are calculated. The final value is then calculated as the ratio of the differences.

Next, I am going to show some examples, with all the values discussed above.

[2_sys.dm_os_performance_counters_ok](#)

Slide 9

Drill Down to Database / File Level

- **sys.dm_io_virtual_file_stats**
 - returns I/O information about *data files* and *log files*
 - parameters
 - `database_ID`
 - NULL = all databases
 - useful function: DB_ID
 - `file_ID`
 - NULL = all files
 - useful function: FILE_IDEX

Related to drill down to database, or File Level, from the query tuning methodology, we have the DMV called sys.dm_io_virtual_file_stats. This one return I/O statistics for data files and log files. It can get all the information of any file in any database.

sys.dm_io_virtual_stats({ database_id | NULL} , { file_id | NULL}),

where

- database_id | NULL –is the ID of the database (database_id is int, with no default). Valid inputs: ID number of a database or NULL. When NULL - all databases in the instance of SQL Server are returned. The built-in function DB_ID can be specified. When using DB_ID without specifying a database name, the compatibility level of the current database must be 90.
- file_id | NULL - ID of the file (file_id is int, with no default). Valid inputs: ID number of a file or NULL. When NULL is specified, all files on the database are returned. The built-in function FILE_IDEX can be specified, and refers to a file in the current database.

Slide 10

Drill Down to Database / File Level

■ sys.dm_io_virtual_file_stats

■ returned table

- database_ID
- file_ID
- sample_ms - # of milliseconds since the computer was started
- num_of_reads - number of reads issued on the file
- num_of_bytes_read - number of bytes read from the file
- io_stall_read_ms - total time users waited for reads issued on the file¹⁰

The table returned contains the following columns:

Column name	Data type	Description
Database_name	Sysname	Database name.
database_id	Smallint	ID of database.
file_id	Smallint	ID of file.
sample_ms	Int	Number of milliseconds since the computer was started. This column can be used to compare different outputs from this function.
num_of_reads	Bigint	Number of reads issued on the file.
num_of_bytes_read	Bigint	Total number of bytes read on this file.
io_stall_read_ms	Bigint	Total time, in milliseconds, that the users waited for reads issued on the file.

Slide 11

Drill Down to Database / File Level

■ `sys.dm_io_virtual_file_stats`

- returned table
 - `num_of_writes` - number of writes
 - `num_of_bytes_written` - total number of bytes written to the file
 - `io_stall_write_ms` - total time users waited for writes to be completed on the file
 - `io_stall` - total time users waited for the completion of I/O operations (ms)
 - `file_handle`

11

Column name	Data type	Description
num_of_writes	Bigint	Number of writes made on this file.
num_of_bytes_written	Bigint	Total number of bytes written to the file.
io_stall_write_ms	Bigint	Total time, in milliseconds, that users waited for writes to be completed on the file.
io_stall	Bigint	Total time, in milliseconds, that users waited for I/O to be completed on the file.
size_on_disk_bytes	Bigint	Number of bytes used on the disk for this file. For sparse files, this number is the actual number of bytes on the disk that are used for database snapshots.
file_handle	Varbinary	Windows file handle for this file.
io_stall_queued_write_ms	Bigint	Total IO latency introduced by IO resource governance for writes. Is not nullable.
io_stall_queued_read_ms	Bigint	Total IO latency introduced by IO resource governance for reads. Is not nullable.

SQL Server performance depends on the I/O subsystem. Unless the database fits into physical memory, SQL Server brings database pages in and out of the buffer pool. This generates substantial I/O traffic. The log records need to be flushed to the disk before a transaction can be declared committed. SQL Server uses TempDB for various purposes (i.e. store intermediate results, to sort, to keep row versions). So, a good I/O subsystem is critical to the performance of SQL Server.

Access to log files is sequential except when a transaction needs to be rolled back while access to data files, including TempDB, is randomly accessed. So, one should have log files on a separate physical disk than data files for better performance. Once an I/O bottleneck is identified, one may need to reconfigure the I/O subsystem.

Now, I am going to show you an example.

[3_sys.dm_io_virtual_file_stats](#)

Slide 12

Drill Down to the Process Level

- a filter on duration or I/O only isolates individual processes (batch / proc / query)
- aggregate performance information by query pattern
 - patterns can be easily identified when using stored procedures
 - if stored procedures are not used:
 - quick and dirty approach: LEFT(query string, n)
 - use a parser to identify the query pattern

12

Another part from the query tuning methodology, refers to drill down to the Process Level. A filter on duration or on I/O only isolates individual processes, like batch, procedure, query. Related to the aggregate performance information by query pattern, the patterns can be easily identified when using stored procedures, and, if the stored procedures are not used, then the approach is quick and dirty (LEFT(query string, n)) and a parser should be used to identify the query pattern.

Slide 13

Indexes

- one of the major factors influencing query performance
 - impact on: filtering, joins, sorting, grouping; blocking and deadlock avoidance, etc.
 - effect on modifications: positive effect (locating the rows); negative effect (cost of modifying the index)
- understanding indexes and their internal mechanisms
 - clustered/nonclustered, single/multicolumn, indexed views, indexes on computed columns, covering scenarios, intersection¹³

The indexes are one of the major factors that influence the query performance. Creating useful indexes is one of the most important ways to achieve better query performance. Useful indexes help in finding data with fewer disk I/O operations and less system resource usage. To create useful indexes, it is important to understand how the data is used, the types of queries and the frequencies that are run, and how the query processor can use indexes to find the data quickly. When decide what indexes to create, examine the critical queries, the performance of which will affect user experience most. Create indexes to specifically aid these queries. After adding an index, rerun the query to see if performance is improved. If it is not, remove the index. As with most performance optimization techniques, there are tradeoffs. For example, with more indexes, **SELECT** queries will potentially run faster. DML (**INSERT**, **UPDATE**, and **DELETE**) operations will slow down significantly because more indexes must be maintained with each operation. If the queries are mostly **SELECT** statements, more indexes can be helpful. If the application performs many DML operations, one should be conservative with the number of indexes created.

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. An index creates a relational index on a table or view.

A **rowstore index** is used to improve query performance, especially when the queries select from specific columns or require values to be sorted in a particular order. This index can be created before there is data in the table.

Unique - Creates a unique index on a table or view. A unique index is one in which no two rows are permitted to have the same index key value. A clustered index on a view must be unique. The Database Engine does not allow creating a unique index on columns that already include duplicate values, whether or not IGNORE_DUP_KEY is set to ON. The Database Engine checks for duplicate values each time data is added by a insert operations. Insert operations that would generate duplicate key values are rolled back, and the Database Engine displays an error message. Columns that are used in a unique index should be set to NOT NULL. This index indicates that the combination of values in the indexed columns must be unique.

Clustered - Creates an index in which the logical order of the key values determines the physical order of the corresponding rows in a table. The bottom, or leaf, level of the clustered index contains the actual data rows of the table. A table or view is allowed one clustered index at a time. A view with a unique clustered index is called an indexed view. Creating a unique clustered index on a view physically materializes the view. A unique clustered index must be created on a view before any other indexes can be defined on the same view. Create the clustered index before creating any nonclustered indexes. Existing nonclustered indexes on tables are rebuilt when a clustered index is created. If CLUSTERED is not specified, a nonclustered index is created.

Nonclustered - Creates an index that specifies the logical ordering of a table. With a nonclustered index, the physical order of the data rows is independent of their indexed order. Each table can have up to 999 nonclustered indexes, regardless of how the indexes are created: either implicitly with PRIMARY KEY and UNIQUE constraints, or explicitly with CREATE INDEX. For indexed views, nonclustered indexes can be created only on a view that has a unique clustered index already defined. If not otherwise specified, the default index type is nonclustered.

Partitioned Indexes - are created and maintained like the partitioned tables, but like ordinary indexes, they are handled as separate database objects. If you are creating an index on a partitioned table, and do not specify a filegroup on which to place the index, the index is partitioned in the same manner as the underlying table. When the index uses the same partition scheme and partitioning column as the table, the index is *aligned* with the table.

Filtered Indexes - is an optimized nonclustered index, suited for queries that select a small percentage of rows from a table. It uses a filter predicate to index a portion of the data in the table. A well-designed filtered index can improve query performance, reduce storage costs, and reduce maintenance costs.

Index Key Size - The maximum size for an index key is 900 bytes for a clustered index and 1,700 bytes for a nonclustered index. The index key of a clustered index cannot contain varchar columns that have existing data in the ROW_OVERFLOW_DATA allocation unit. Nonclustered indexes can include non-key columns in the leaf level of the index. These columns are not considered by the Database Engine when calculating the index key size.

Indexing on columns used in the WHERE clause of your critical queries frequently improves performance. However, this depends on how selective the index is. Selectivity is the ratio of qualifying rows to total rows. If the ratio is low, the index is highly selective. It can get rid of most of the rows and greatly reduce the size of the result set. It is therefore a useful index to create. By contrast, an index that is not selective is not as useful. A unique index has the greatest selectivity. Only one row can match, which makes it most helpful for queries that intend to return exactly one row. For example, an index on a unique ID column will help on finding a particular row quickly. Evaluate the selectivity of an index by running the *sp_show_statistics* stored procedures on SQL Server Compact tables.

Multiple-column indexes are natural extensions of single-column indexes. Multiple-column indexes are useful for evaluating filter expressions that match a prefix set of key columns. When it is created a multiple-column index, should be put it the most selective columns leftmost in the key. This makes the index more selective when matching several expressions.

It is recommended to create indexes on primary keys. It is frequently useful to also create indexes on foreign keys. This is because primary keys and foreign keys are frequently used to join tables. Indexes on these keys lets the optimizer consider more efficient index join algorithms. If the query joins tables by using other columns, it is frequently helpful to create indexes on those columns for the same reason. When primary key and foreign key constraints are created, SQL Server Compact automatically creates indexes for them and takes advantage of them when optimizing queries. Remember to keep primary keys and foreign keys small. Joins run faster this way.

Indexes can be used to speed up the evaluation of certain types of filter clauses. Although all filter clauses reduce the final result set of a query, some can also help reduce the amount of data that must be scanned.

Indexes can be created on computed columns. These ones can have the property **PERSISTED** (the Database Engine stores the computed values in the table, and updates them when any other columns on which the computed column depends are updated). The Database Engine uses these persisted values when it creates an index on the column, and when the index is referenced in a query. To index a computed column, the computed column must deterministic and precise.

Non-key columns (or included columns), can be added to the leaf level of a nonclustered index to improve query performance by covering the query. This allows the query optimizer to locate all the required information from an index scan; the table or clustered index data is not accessed. If an index exists, the optimizer evaluates the index by calculating how many rows are returned. It then estimates the cost of finding the qualifying rows by using the index. It will choose indexed access if it has lower cost than table scan.

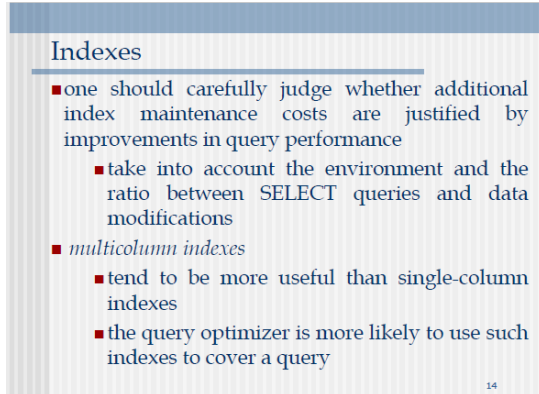
So, the indexes have impact on: filtering, joins, sorting, grouping, also, on blocking and deadlock avoidance. The indexes have positive effects on modifications, when locating the rows, and have negative effect, related to the cost of modifying the index.

It is also very important to understand the indexes and the internal mechanisms for all types of indexes, like: clustered, nonclustered, single, multicolumn, indexed views, indexed on computed columns, covering scenarios, intersection.

One can define indexes on computed columns as long as the following requirements are met:

- Ownership requirements - All function references in the computed column must have the same owner as the table.
- Determinism requirements – expressions are deterministic if they always return the same result for a specified set of inputs. The *computed_column_expression* must be deterministic (All functions (user-defined, built-in functions) that are referenced by the expression are deterministic and precise and also all columns that are referenced in the expression come from the table that contains the computed column; No column reference pulls data from multiple rows; The *computed_column_expression* has no system data access or user data access;). Any computed column that contains a common language runtime (CLR) expression must be deterministic and marked **PERSISTED** before the column can be indexed.
- Precision requirements
- Data type requirements
- SET option requirements

Slide 14



Slide 14 is titled "Indexes". It contains a bulleted list with the following items:

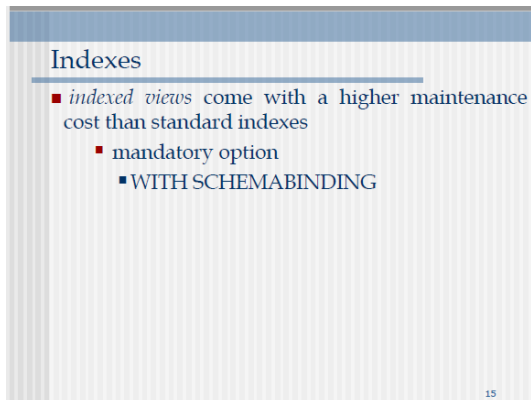
- one should carefully judge whether additional index maintenance costs are justified by improvements in query performance
 - take into account the environment and the ratio between SELECT queries and data modifications
- *multicolumn indexes*
 - tend to be more useful than single-column indexes
 - the query optimizer is more likely to use such indexes to cover a query

The slide number "14" is visible in the bottom right corner.

Also, we should pay attention whether additional index maintenance costs are justified by improvements in query performance, and take into account the environment and the ratio between SELECT queries and data modifications.

The multicolumn indexes tend to be more useful than single-column indexes and the query optimizer is more likely to use such indexes to cover a query.

Slide 15



Slide 15 is titled "Indexes". It contains a bulleted list with the following items:

- *indexed views* come with a higher maintenance cost than standard indexes
 - mandatory option
 - WITH SCHEMABINDING

The slide number "15" is visible in the bottom right corner.

The first index created on a view must be a unique clustered index. After the unique clustered index has been created, there can be more nonclustered indexes. Creating a unique clustered index on a view improves query performance because the view is stored in the database in the same way a table with a clustered index is stored. The query optimizer may use indexed views to speed up the query execution. The view does not have to be referenced in the query for the optimizer to consider that view for a substitution. To create an indexed view must:

- Verify the SET options are correct for all existing tables that will be referenced in the view.
- Verify that the SET options for the session are set correctly before you create any tables and the view.
- Verify that the view definition is deterministic.
- Create the view by using the WITH SCHEMABINDING option.
- Create the unique clustered index on the view.

When SCHEMABINDING is specified, the base table/s cannot be modified in a way that would affect the view definition. In SCHEMABINDING, the *select_statement* must include the two-part names (*schema.object*) of tables, views, or user-defined functions that are referenced. All referenced objects must be in the same database. Views or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped unless that view is dropped or changed so that it no

longer has schema binding. Otherwise, the Database Engine raises an error. SCHEMABINDING cannot be used in Stored Procedures. So, by including the SCHEMABINDING clause the view/function are protected from unexpected changes to the tables that underneath them.

Next, I am going to show some examples relative to the indexes.

Clustered
indexes_5

Slide 16

Tools to Analyze Query Performance

- graphical execution plan
- STATISTICS IO - scan count, logical reads, physical reads, *read-ahead* reads
- STATISTICS TIME - duration and net CPU time
- SHOWPLAN_TEXT - SQL Server returns detailed information about how the statements are executed
- SHOWPLAN_ALL - SQL Server returns detailed information about how the statements are executed, provides estimates of the resource requirements
- STATISTICS PROFILE - actual plan

16

The following part of this seminar, it is dedicated to the tools used to analyze the query performance. Here, we discuss about instruments like: graphical execution plan, STATISTICS IO, STATISTICS TIME, SHOWPLAN_TEXT, SHOWPLAN_ALL, STATISTICS PROFILE, STATISTICS XML, SHOWPLAN_XML. We are going to analyze each of ones and we will see examples.

Microsoft SQL Server provides a set of tools for monitoring events in SQL Server and for tuning the physical database design. The choice of tool depends on the type of monitoring or tuning to be done and the particular events to be monitored.

Tool	Description
sp_trace_setfilter (Transact-SQL)	SQL Server Profiler tracks engine process events (i.e. start of a batch or a transaction, enable to monitor server and database activity (for example, deadlocks, fatal errors, or login activity)). One can capture SQL Server Profiler data to a SQL Server table or a file for later analysis, and can replay the events captured on SQL Server step by step, to see exactly what happened.
SQL Server Distributed Replay	Microsoft SQL Server Distributed Replay can use multiple computers to replay trace data, simulating a mission-critical workload.
Monitor Resource Usage (System Monitor)	System Monitor primarily tracks resource usage (i.e. the number of buffer manager page requests in use) enabling to monitor server performance and activity using predefined objects and counters or user-defined counters to monitor events. System Monitor (Performance Monitor in Microsoft Windows NT 4.0) collects counts and rates rather than data about the events (i.e. memory usage, number of active transactions, number of blocked locks, or CPU activity). One can set thresholds on specific counters to generate alerts that notify operators. System Monitor works on Microsoft Windows Server and Windows operating systems. SQL Server Profiler monitors Database Engine events and System Monitor monitors resource usage associated with server processes.
Open Activity Monitor (SQL Server Management Studio)	The Activity Monitor in SQL Server Management Studio is useful for ad hoc views of current activity and graphically displays information about: Processes running on

	an instance of SQL Server, Blocked processes, Locks, User activity.
Live Query Statistics	Displays real-time statistics about query execution steps. This data is available while the query is executing, so these execution statistics are extremely useful for debugging query performance issues.
SQL Trace	Transact-SQL stored procedures that create, filter, and define tracing: <code>sp_trace_create</code> , <code>sp_trace_generateevent</code> , <code>sp_trace_setevent</code> , <code>sp_trace_setfilter</code> , <code>sp_trace_setstatus</code> .
Error Logs	The Windows application event log provides an overall picture of events occurring on the Windows Server and Windows operating systems, events in SQL Server, SQL Server Agent, and full-text search. It contains information about events in SQL Server that is not available elsewhere. One can use the information in the error log to troubleshoot SQL Server-related problems.
System Stored Procedures (Transact-SQL)	The SQL Server system stored procedures provide an alternative for many monitoring tasks: <code>sp_who</code> - Reports snapshot information about current SQL Server users and processes, the currently executing statement and whether the statement is blocked. <code>sp_lock</code> - Reports snapshot information about locks, including the object ID, index ID, type of lock, and type or resource to which the lock applies. <code>sp_spaceused</code> - Displays an estimate of the current amount of disk space used by a table (or a whole database). <code>sp_monitor</code> - Displays statistics, including CPU usage, I/O usage, and the amount of time idle since <code>sp_monitor</code> was last executed.
DBCC (Transact-SQL)	DBCC (Database Console Command) statements enable you to check performance statistics and the logical and physical consistency of a database.
Built-in Functions (Transact-SQL)	Built-in functions display snapshot statistics about SQL Server activity since the server was started; these statistics are stored in predefined SQL Server counters. I.e. <code>@@CPU_BUSY</code> contains the amount of time the CPU has been executing SQL Server code; <code>@@CONNECTIONS</code> contains the number of SQL Server connections or attempted connections; and <code>@@PACKET_ERRORS</code> contains the number of network packets occurring on SQL Server connections.
Trace Flags (Transact-SQL)	Trace flags display information about a specific activity within the server and are used to diagnose problems or performance issues (for example, deadlock chains).
Database Engine Tuning Advisor	Database Engine Tuning Advisor analyzes the performance effects of Transact-SQL statements executed against databases you want to tune. Database Engine Tuning Advisor provides recommendations to add, remove, or modify indexes, indexed views, and partitioning.

The choice of a monitoring tool depends on the event or activity to be monitored.

Now, let us come back to the tools specified on this slide.

For the *graphical execution plan*, we mean the execution plans that are generated after the Transact-SQL queries or batches execute. Because of this, an actual execution plan contains runtime information, such as actual resource usage metrics and runtime warnings (if any). The execution plan that is generated, displays the actual query execution plan that the SQL Server Database Engine used to execute the queries. For this feature, users must have permissions to execute the Transact-SQL queries for which a graphical execution plan is being generated, and they must be granted the `SHOWPLAN` permission for all databases referenced by the query. Beside this actual execution plan, we can also specify the estimated execution plan, that estimates the work that SQL Server is expected to perform to get the data.

The *STATISTICS IO* allows the scan count, the logical reads, physical reads and read-ahead reads. SQL Server's *STATISTICS IO* reporting is a great tool to help performance tune queries. The goal of performance tuning is to make the query run faster. One way to get a faster query is to reduce the amount of data that a query is processing. The *STATISTICS IO* output helps with performance tuning because the data it shows acts as a measuring stick for the performance tuning changes and it provides a good way of isolating the query changes.

STATISTICS IO provides detailed information about the impact that the query has on SQL Server, telling the number of logical reads (including LOB), physical reads (including read-ahead and LOB), and how many times a table was scanned. This information helps you to establish whether or not the choices made by the optimizer are as efficient as possible at the time.

STATISTICS IO can be set as an option when execute a query. A message is sent via the connection that made a query, telling the cost of the query in terms of the actual number of physical reads from the disk and logical reads from memory, by the query. The *STATISTICS IO* output contains:

- **Logical reads:** The number of 8kB pages SQL Server had to read from the buffer cache (memory) in order to process and return the results of the query. The more pages that need to be read, the slower will be the query.
- **Worktables/Workfiles:** These are temporary objects that SQL Server creates in tempdb in order to process query results.
- **Lob Logical Reads:** The number of large objects (e.g. varchar(max)) SQL is having to read.

The tool called *STATISTICS TIME* gives us the duration and net CPU time. It is displaying the number of milliseconds require to parse, compile and execute each statement.

When *SET STATISTICS TIME* is ON, the time statistics for a statement are displayed. When OFF, the time statistics are not displayed. The setting of *SET STATISTICS TIME* is set at execute or run time and not at parse time. Microsoft SQL Server is unable to provide accurate statistics in fiber mode, which is activated when the lightweight pooling configuration option is enabled. The **cpu** column in the **sysprocesses** table is only updated when a query executes with *SET STATISTICS TIME* ON. When *SET STATISTICS TIME* is OFF, 0 is returned. ON and OFF settings also affect the CPU column in the Process Info View for Current Activity in SQL Server Management Studio. To use *SET STATISTICS TIME*, users must have the appropriate permissions to execute the Transact-SQL statement. The *SHOWPLAN* permission is not required.

At the *SHOWPLAN_TEXT*, SQL Server returns detailed information about how the statements are executed. This tool causes that SQL Server not execute Transact-SQL statements. Instead, SQL Server returns detailed information about how the statements are executed. The setting of *SET SHOWPLAN_TEXT* is set at execute or run time and not at parse time.

When *SET SHOWPLAN_TEXT* is ON, SQL Server returns execution information for each Transact-SQL statement without executing it. After this option is set ON, execution plan information about all subsequent SQL Server statements is returned until the option is set OFF. For example, if a *CREATE TABLE* statement is executed while *SET SHOWPLAN_TEXT* is ON, SQL Server returns an error message from a subsequent *SELECT* statement involving that same table informing the user that the specified table does not exist. Therefore, subsequent references to this table fail. When *SET SHOWPLAN_TEXT* is OFF, SQL Server executes statements without generating a report with execution plan information.

SET SHOWPLAN_TEXT is intended to return readable output for Microsoft Win32 command prompt applications such as the **sqlcmd** utility. SET SHOWPLAN_ALL returns more detailed output intended to be used with programs designed to handle its output.

SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL cannot be specified in a stored procedure. They must be the only statements in a batch.

SET SHOWPLAN_TEXT returns information as a set of rows that form a hierarchical tree representing the steps taken by the SQL Server query processor as it executes each statement. Each statement reflected in the output contains a single row with the text of the statement, followed by several rows with the details of the execution steps.

Corresponding to the tool *SHOWPLAN_ALL*, SQL Server returns detailed information about how the statements are executed, and also provides estimates of the resource requirements.

Causes Microsoft SQL Server not to execute Transact-SQL statements. Instead, SQL Server returns detailed information about how the statements are executed and provides estimates of the resource requirements for the statements. The setting of SET SHOWPLAN_ALL is set at execute or run time and not at parse time.

When SET SHOWPLAN_ALL is ON, SQL Server returns execution information for each statement without executing it, and Transact-SQL statements are not executed. After this option is set ON, information about all subsequent Transact-SQL statements are returned until the option is set OFF. For example, if a CREATE TABLE statement is executed while SET SHOWPLAN_ALL is ON, SQL Server returns an error message from a subsequent SELECT statement involving that same table, informing users that the specified table does not exist. Therefore, subsequent references to this table fail. When SET SHOWPLAN_ALL is OFF, SQL Server executes the statements without generating a report. SET SHOWPLAN_ALL is intended to be used by applications written to handle its output. Use SET SHOWPLAN_TEXT to return readable output for Microsoft Win32 command prompt applications, such as the **osql** utility.

SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL cannot be specified inside a stored procedure; they must be the only statements in a batch.

SET SHOWPLAN_ALL returns information as a set of rows that form a hierarchical tree representing the steps taken by the SQL Server query processor as it executes each statement. Each statement reflected in the output contains a single row with the text of the statement, followed by several rows with the details of the execution steps.

The tool *STATISTICS PROFILE* refers to the actual plan.

Displays the profile information for a statement. STATISTICS PROFILE works for ad hoc queries, views, and stored procedures. When STATISTICS PROFILE is ON, each executed query returns its regular result set, followed by an additional result set that shows a profile of the query execution.

The additional result set contains the SHOWPLAN_ALL columns for the query and these additional columns; rows – indicates the actual number of rows produced by each operator; and executed – indicates the Number of times the operator has been executed.

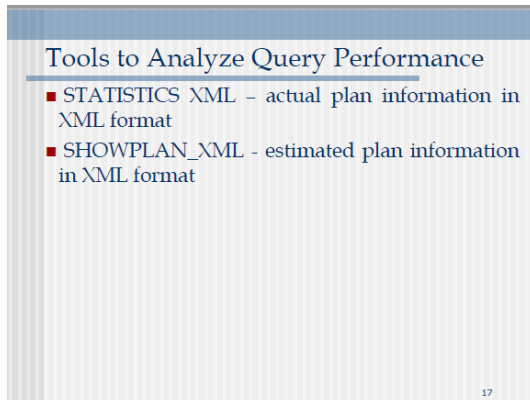
To use SET STATISTICS PROFILE and view the output, users must have the following permissions:

- Appropriate permissions to execute the Transact-SQL statements.
- SHOWPLAN permission on all databases containing objects that are referenced by the Transact-SQL statements.

For Transact-SQL statements that do not produce STATISTICS PROFILE result sets, only the appropriate permissions to execute the Transact-SQL statements are required. For Transact-SQL

statements that do produce STATISTICS PROFILE result sets, checks for both the Transact-SQL statement execution permission and the SHOWPLAN permission must succeed, or the Transact-SQL statement execution is aborted and no Showplan information is generated.

Slide 17



Tools to Analyze Query Performance

- STATISTICS XML - actual plan information in XML format
- SHOWPLAN_XML - estimated plan information in XML format

17

The tool *STATISTICS XML*, give us the actual plan information in XML format.

SET STATISTICS xml {on|off} - is set at execute / run time, not at parse time

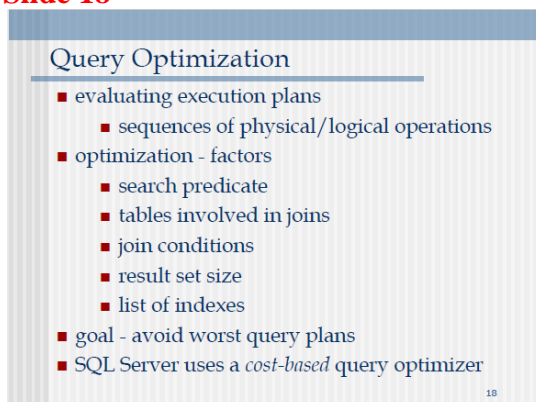
When SET STATISTICS XML is ON, SQL Server returns execution information for each statement after executing it and until the option is set to OFF. It returns output as nvarchar(max) for applications, as a set of XML documents. Each statement after the SET STATISTICS XML ON statement is reflected in the output by a single document (with the details of the execution steps). The output shows run-time information (i.e. the costs, accessed indexes, and types of operations performed, join order, the number of times a physical operation is performed, the number of rows each physical operator produced).

Te tool *SHOWPLAN_XML*, returns the estimated plan information in XML format.

Next, I am going to show you a short example.

[5_tools_analyze_query_performance](#)

Slide 18



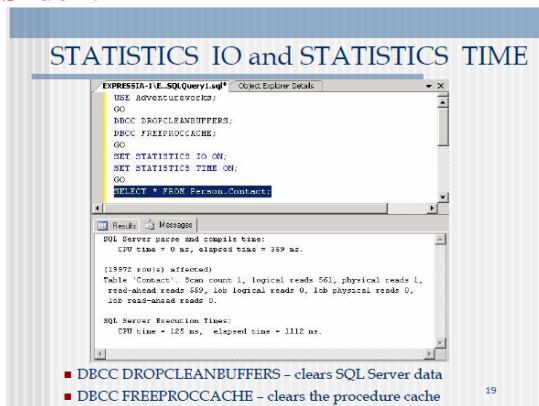
Query Optimization

- evaluating execution plans
 - sequences of physical/logical operations
- optimization - factors
 - search predicate
 - tables involved in joins
 - join conditions
 - result set size
 - list of indexes
- goal - avoid worst query plans
- SQL Server uses a *cost-based* query optimizer

18

The Query Optimization refers to the evaluation of the execution plans and to the factors that are used in optimization. The execution plans are evaluated, we consider the sequences of physical and also logical operations. The factors used I optimization are: the search predicate, the tables involved in joins, the join conditions, the results set size and the list of indexes. IN query optimization, the goal is to avoid worst query plans. Related to SQL Server, for the query optimizer, there is a cost-based optimizer.

Slide 19



All the slides that follows will show us, examples related to the tools used to analyze the query performance.

We start with the STATISTICS IO and STATISTICS TIME, and we introduce two procedures:

- DBCC DROPCLEANBUFFERS – that clears SQL Server data
- DBCC FREEPROCCACHE – that clears the procedure cache

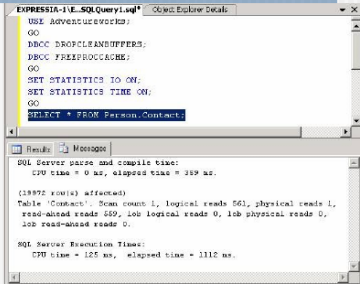
DBCC DROPCLEANBUFFERS removes all clean buffers from the buffer pool, and columnstore objects from the columnstore object pool. Use DBCC DROPCLEANBUFFERS to test queries with a cold buffer cache without shutting down and restarting the server. To drop clean buffers from the buffer pool and columnstore objects from the columnstore object pool, first use CHECKPOINT to produce a cold buffer cache. This forces all dirty pages for the current database to be written to disk and cleans the buffers. After you do this, you can issue DBCC DROPCLEANBUFFERS command to remove all buffers from the buffer pool.

DBCC FREEPROCCACHE removes all elements from the plan cache, removes a specific plan from the plan cache by specifying a plan handle or SQL handle, or removes all cache entries associated with a specified resource pool. DBCC FREEPROCCACHE does not clear the execution statistics for natively compiled stored procedures. The procedure cache does not contain information about natively compiled stored procedures. Any execution statistics collected from procedure executions will appear in the execution statistics DMVs: sys.dm_exec_procedure_stats (Transact-SQL) and sys.dm_exec_query_plan (Transact-SQL). Use DBCC FREEPROCCACHE to clear the plan cache carefully. Clearing the procedure (plan) cache causes all plans to be evicted, and incoming query executions will compile a new plan, instead of reusing any previously cached plan.

After, we execute these 2 procedures, we set the STATISTICS IO and the STATISTICS TIME, ON, and we execute a select from a table, and we obtain the following result.

Slide 20

STATISTICS IO and STATISTICS TIME



```

USE Adventureworks;
GO
DBCC DROPCLEANUPPER;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
    
```

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 219 ms.

(19972 row(s) affected)
Table 'Contact': Scan count 1, logical reads 561, physical reads 1,
read-ahead reads 559, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 125 ms, elapsed time = 1112 ms.

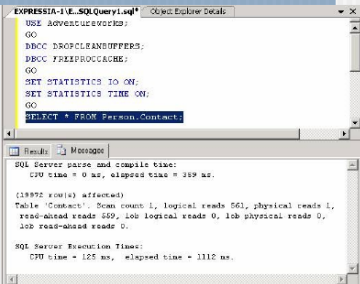
- CPU time - CPU resources used to execute the query
- elapsed time - how long the query took to execute

20

This result includes the CPU time used, that represent the CPU resources used to execute the query; and the elapsed time, or, how long the query took to execute.

Slide 21

STATISTICS IO and STATISTICS TIME



```

USE Adventureworks;
GO
DBCC DROPCLEANUPPER;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
    
```

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 219 ms.

(19972 row(s) affected)
Table 'Contact': Scan count 1, logical reads 561, physical reads 1,
read-ahead reads 559, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 125 ms, elapsed time = 1112 ms.

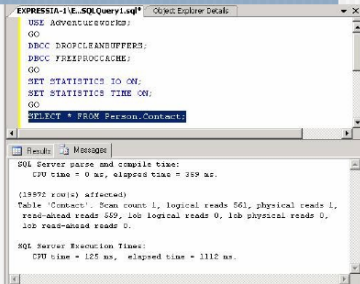
- physical reads - number of pages read from the disk
- read-ahead reads - number of pages placed in the cache for the query

21

The result, also, returns us, the physical reads, or the number of pages read from the disk, and the read-ahead reads, or the number of pages placed in the cache for the query.

Slide 22

STATISTICS IO and STATISTICS TIME



```

USE Adventureworks;
GO
DBCC DROPCLEANUPPER;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
    
```

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 219 ms.

(19972 row(s) affected)
Table 'Contact': Scan count 1, logical reads 561, physical reads 1,
read-ahead reads 559, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 125 ms, elapsed time = 1112 ms.

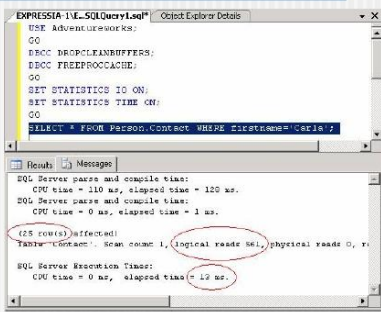
- scan count - how many times have the tables been accessed
- logical reads - number of pages read from the data cache

22

Another part of the result refers to the scan count, that returns how many times have the tables been accessed, and, to the logical reads, or the number of pages read from the data cache.

Slide 23

STATISTICS IO and STATISTICS TIME



```
USE Adventureworks;
GO
DBCC DROPCLANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact WHERE FirstName='Gail';
```

Results: Messages

SQL Server parse and compile time:
CPU time = 110 ms, elapsed time = 120 ms.

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 1 ms.

(25 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 661, physical reads 0, ...

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 12 ms.

Here, one can see some of the results circled.

Now, I am going to show you this example in practice.

6_statistics (PART 1)

Slide 24

STATISTICS IO and STATISTICS TIME

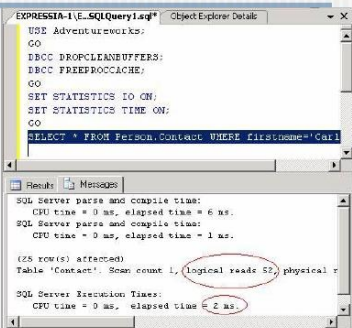
```
USE AdventureWorks
GO
CREATE NONCLUSTERED INDEX IDX_FirstName
ON Person.Contact(FirstName ASC)
GO
```

Next, we want to introduce a WHERE clause to our query. First it is important to check the query without this index, and then, create the index, and check the same query, with the index used in the WHERE clause. From here we can extract the importance of using indexes on the columns involved specific in queries.

So, we create an index on the Firstname on the table Person.Contact and we execute the following query:

Slide 25

STATISTICS IO and STATISTICS TIME



```
USE Adventureworks;
GO
DBCC DROPCLANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact WHERE FirstName='Gail';
```

Results: Messages

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 6 ms.

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 1 ms.

(25 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 62, physical reads 0, ...

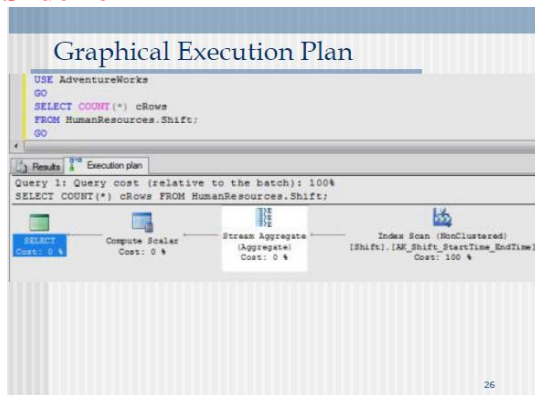
SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 2 ms.

This is the query.

Now, let me show you, the example, in practice.

6_statistics (PART 2)

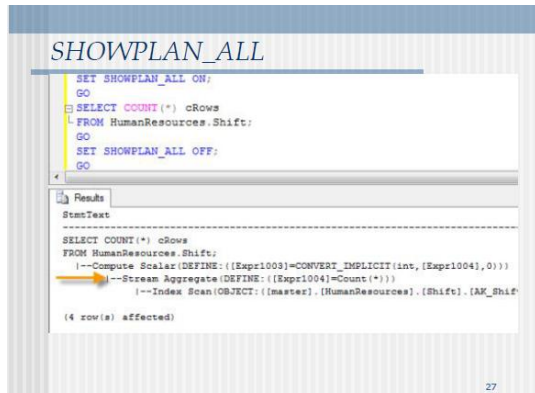
Slide 26



Next, we will see an example, related to the graphical execution plan, example in which we will need to use the Include Live Query Statistics.

6_statistics (PART 3)

Slide 27

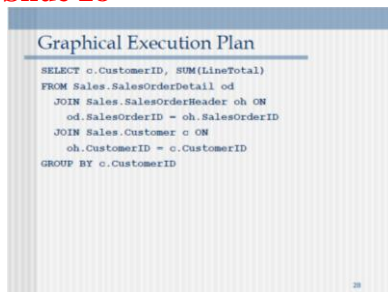


Now, for the same query, we will check also, the SHOWPLAN_ALL. First, we need to set the SHOWPLAN_ALL to ON, then execute the query, and then, to set the SHOWPLAN_ALL to OFF.

Let's see, now, the example

6_statistics (PART 4)

Slide 28



Graphical Execution Plan

CREATE INDEX IX_OrderDetail_OrderID_TotalLine
ON Sales.SalesOrderDetail (SalesOrderID)
INCLUDE (LineTotal)

Hash Join (HJ)
Hash Match (HM)
Hash Build (HB)
Table Scan (TS)
SalesOrderDetail
IX_OrderDetail_OrderID_TotalLine
Table Scan (TS)
SalesOrderDetail

33

Now, let me show you, these examples, in practice.

Bibliografy:

<https://developer.rackspace.com/blog/understanding-a-sql-server-execution-plan/>

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-statistics-time-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-showplan-all-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-dropcleanbuffers-transact-sql?view=sql-server-ver15>

<https://docs.microsoft.com/en-us/sql/t-sql/database-console-commands/dbcc-freeproccache-transact-sql?view=sql-server-ver15>