Troubleshoot Database Concurrency in SQL Server with sp_locks

BY YANIV ETROGI

General

Database concurrency can be defined as the number of users that can work at a given database at the same time while not effecting or interfering with eachother's work. The greater the number of users working at the same time the higher concurrency is.

SQL Server holds Locks on user data to protect data integrity and this is required in every database where data gets modified. Just to emphasize, be aware that if a database is set in a READ_ONLY state SQL Server does not hold locks on data being read since it cannot be modified and so there is no data that needs to be protected, therefore SQL Server only marks the objects being accessed with an Intense Share (IS) lock to prevent a DDL statement such as dropping a table while it's data is being read.

Locks on user data exist at any environment where data gets modified (the database is set in READ_WRITE state as opposed to the above example) and this is absolutely OK. SQL Server holds the required locks to protect data integrity and releases the locks as soon as they are no longer required.

Short terms locks have no negative impact and they are part of the normal work that SQL Server carries out, problems arise when locks duration becomes long. i.e. a few seconds and this is when sp_locks step in.

sp_locks

sp_locks is a useful tool that can help you in detecting and troubleshooting blocking and concurrency scenarios.

Unlike *sp_helpindex2* which retrieves information related to the context of the current database *sp_locks*retrieves server level information and therefore does not need to be marked as a system stored procedure. However, by using the prefix *sp_* we take advantage of the fact that sql server first searches the master database for any object prefixed with *sp_* allowing us to call the procedure from any database without the need to use a three part name (i.e. database.schema.object) making it's usage more convenience.

The procedure retunes 3 result sets at the most and this is controlled by the @Mode input parameter.

The first result set returns information about the *Lead Blocker* process if exists and is aimed towards a situation where there is a high blocking activity resulting in a *Blocking Chain*.

A blocking chain is the situation where there are many processes that are blocking other processes that are now being blocked and in many cases requires a manual interference to get out of the situation, typically issuing a kill command to terminate the process that is the Lead Blocker.

The Lead Blocker is a nickname we call the process that is the root cause, the initiator, that's the process that 'started' the blocking chain. i.e. a process that blocks a second process that now since it is being blocked does not release it's granted locks thus becoming a blocker itself and this can go on and on while other processes join the party resulting in a chain of blocks.

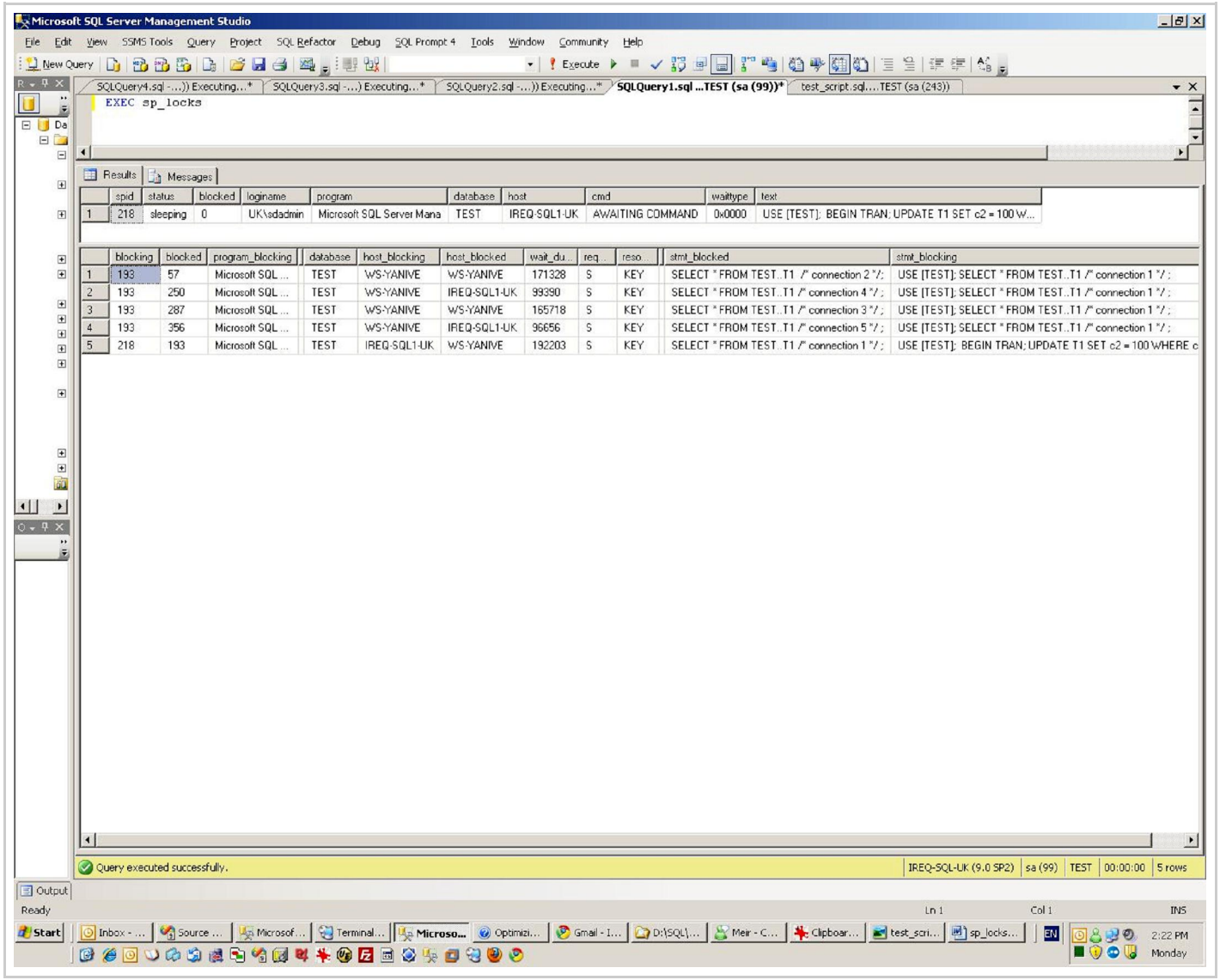
In a such a scenario database concurrency is negatively effected and this of course gets reflected in the application's response time so at the moment you become aware of the situation what you really want to do is handle it as soon as you can to minimize the effect on your system.

This result set will return a single row in most cases but under an extreme blocking scenario it is not unlikely that you will see more than a single row returned (meaning we have more than one Lead Blocker or more than one Blocking Chain).

In the test script scenario's screen shot that I captured the lead blocker is the process issuing the UPDATE command (spid 218). This process is blocking the first process (spid 193) that issues the SELECT statement (/* connection 1 */) and every additional SELECT statement issued there after is blocked by that first SELECT statement that is blocked by the process that issued the UPDATE command.

Terminating the connection that issued the UPDATE command will let all the blocked (waiting) processes to proceed with their tasks and resolve the blocking scenario and get rid of the blocking chain. So in this case the process issuing the UPDATE command is the lead blocker.

Note that if you terminate any other process you don't resolve the blocking scenario and the blocking chain remains.



The second result set retrieves information about processes blocking and processes being blocked by querying three key DMVs: *sys.dm_os_waiting_tasks*, *sys.dm_tran_locks* and *sys.sysprocesses*.

We query *sys.dm_os_waiting_tasks* because that is actually what we are intrested in – tasks (processes) that are now waiting, waiting to be granted locks on resources in our case. The cool part is that this DMV can be joined to the sys.dm_tran_locks on the resource_address column which is the memory address of the resource for which the task is waiting for and on the other side of the join condition we have the *lock_owner_address*column in sys.dm_tran_locks which is the memory address of the internal data structure that is used to track the lock request by the LOCK_MANAGER module in

SQL Server.

These two DMVs expose valuable information when it comes to locking and blocking. Next we add*sys.sysprocesses* to get higher level information such as the program and host blocking. In order to retrieve information for both processes, the blocker process and the process being blocked we join *sys.sysprocesses*twice while one instance represents the blocker and the second instance represents the process being blocked.

This is achieved by joining one instance of sys.sysprocesses as p1 with sys.dm_os_waiting_tasks on the session_id column while the second instance of of sys.sysprocesses is joined as p2 on sys.dm_*os_waiting_tasks*'s *blocking_session_id* column.

The @*Wait_Duration_ms* input parameter allows you to limit the rows returned to those processes who's wait time exceeds the number (an integer representing the time duration in milliseconds) you provide. This is very useful because typically you have no interest in short term duration (i.e. two or three seconds) waits (locks) that get released and have no negative impact on the system.

The third result set retrieves information from *sys.dm_exec_requests* about active executions currently running. This result set becomes useful when there is no blocking activity taking place and so no rows are returned by the first two result sets but you still want to see what executions are now running on the server.

Continues…