**DEADLOCKS**

* Occurs when 2 processes are competing for exclusive access to a resource
* Essentially a stand off where neither process can proceed
* Resolved by killing one of the processes
* SQL Server automatically detects deadlocks and selects one of the process as the victim, kills it and thus allows the other process to continue
* Deadlocks do not only occur on locks, from SQL Server 2012 onward, deadlocks can also happen:

1. with memory
2. MARS (Multiple Active Result Sets) resources
3. worker threads
4. resources related to parallel query execution

**Deadlock Types**

* *A Cycle Deadlock*: Occurs when Process A which is holding a lock on Resource X is waiting to obtain an exclusive lock on Resource Y and at the same time Process B is locking Resource X in order to obtain an exclusive lock on Resource X. Hence both locks are in contention
* *Conversion locks deadlock*: Occurs when a thread tries to convert a lock from one type to another but can’t because another thread is also holding a shared lock on the same resource

**Handling Deadlocks**

* SQL lock manager (LOCK\_MONITOR thread) automatically searches for deadlocks every 5secs
* Chooses a victim and kills terminates the transaction, rolls it back and kills the connection to release all resources
* Deadlock search frequency is automatically adjusted as required (default is 5 secs)
* Victim is chosen based on the deadlock priority setting of a transaction

**SETTINGS**

SET DEADLOCK\_PRIORITY { LOW | NORMAL | HIGH, <numeric-priority> | @deadlock\_var | @deadlock\_intvar }

1. LOW – the current session will be the deadlock victim if other sessions have Normal/High priority or to an integer value > -5. Or, if the other session has priority LOW and an integer session = -5.
2. NORMAL – if the other session is set to Normal/High or to an integer value = 0 (but not if the settings are Low and integer value > 0). Normal is the default priority
3. HIGH – if other sessions have integer value > 5 or if the other session is set to High or to an integer value = 5.
4. <numeric-priority> - range (-10 to 10) providing 21 levels of deadlock priority. Low maps to -5, Normal maps to 0 and High maps to 5.
5. @deadlock-var – character variable specifying the priority ‘LOW’, ‘NORMAL’ or ‘HIGH’
6. @deadlock\_intvar – Integer value in the range (-10 to 10)

Notes:

1. SET DEADLOCK\_PRIORITY is set at execute time or run time not at parse time.
2. If both sessions have the same priority SQL chooses the session which is least expensive to roll back. The cost is determined by log bytes written to that point in each transaction, (“Log Used” value in the deadlock graph).

**Minimise Deadlocks**

* Always try to hold locks for as short a period as possible
* Always access resources in the same order
* Ensure that you don’t have to wait on user input in the middle of a transaction, get all required information and then submit the transaction
* Try to limit lock escalation, by using hints such as NOLOCK (on SELECT statements) and HOLDLOCK and UPDLOCK on transaction statements
* Enclose transactions in TRY/CATCH block and add retry logic
* Add a new covering non-clustered index to provide another way for SQL to read data without requiring access to the underlying table (additional overhead)
* Use READ COMMITTED SNAPSHOT or SNAPSHOT which avoid most blocking problems without the risk of dirty reads (both require resources on *tempdb*)

**Monitor Deadlocks**

* Capture and analyse deadlock graphs using SQL Server Profiler or Extended events
* A deadlock graph is an XML description of a deadlock
* Also Trace Flags 1204 and 1222, enabled using DBCC TRACEON(1204,1222,-1)
* When using profiler to capture a deadlock graph, the trace must be configured before the deadlock occurs using the following events:

1. Deadlock graph
2. Lock: Deadlock
3. Lock: Deadlock Chain