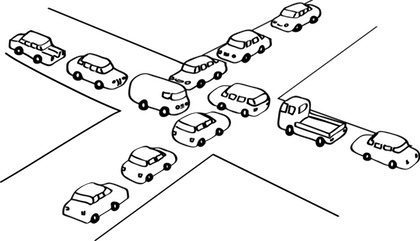
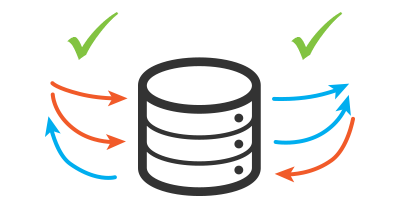
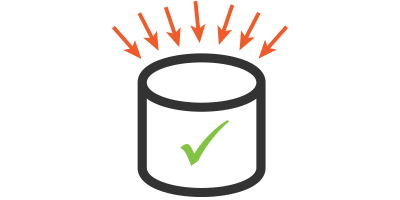
**Concurrency Control** **And Locking**





**Concurrency and Locking – Interview Questions with answers**

**1. What are the concurrent problems occur in accessing database?**

**Ans:**

Tr – Transaction

R – Resource

**Uncommitted dependency/dirty reads:**

Tr1 Updates Data R1

Tr2 Reads Data R1

Tr1 Rollback the Update Operation R1

Now Tr2 has the inconsistent data or wrong data.

**Inconsistent Analysis/non-repeatable reads**

Tr1 Reads Data R1

Tr2 Updates Data R1

Tr1 Again Reads the same data R1

Wrong match between first Tr1 and Second time Tr1

**Phantom reads (via insert/delete)**

Tr1 Reads Data (Result Contains 10 Records R1

Tr2 Insert/Delete Data (insert 6 new delete 1 Record) R1

Tr1 Again reads the same data R1

In Second time Tr1 we found 6 New Records and we can’t find a record which retrieves in first time…..

**2. What isolation levels will provide completely read-consistent views of a database to all transactions?**

**Ans:**

**SQL Server 2000:** Only the SERIALIZABLE isolation level will provide a completely read-consistent view of a database to a given transaction. In any of the other isolation levels, you could perceive some/all of the following, depending on the isolation level running in:

* + Uncommitted dependency/dirty reads
  + Inconsistent Analysis/non-repeatable reads
  + Phantom reads (via insert/delete)

**SQL Server 2005 and above:** Both the SERIALIZABLE and SNAPSHOT isolation levels will provide a completely read-consistent view of a database to a given transaction. In any of the other isolation levels, you could perceive some/all of the following, depending on the isolation level running in:

* + Uncommitted dependency/dirty reads
  + Inconsistent Analysis/non-repeatable reads
  + Phantom reads (via insert/delete)

**3. Within the READ\_COMMITTED isolation level, during a read operation how long are locks held/retained for?**

**Ans:**

When SQL Server executes a statement at the read committed isolation level, it acquires short lived share locks on a row by row basis. The duration of these share locks is just long enough to read and process each row; the server generally releases each lock before proceeding to the next row. Thus, if you run a simple select statement under read committed and check for locks, you will typically see at most a single row lock at a time. The sole purpose of these locks is to ensure that the statement only reads and returns committed data. The locks work because updates always acquire an exclusive lock which blocks any readers trying to acquire a share lock.

**4. Within the REPEATABLE\_READ and SERIALIZABLE isolation levels, during a read operation and assuming row-level locking, how long are locks held/retained for?**

**Ans:**

Within either of these isolation levels, locks are held for the duration of the transaction, unlike within the READ\_COMMITTED isolation level as noted above.

**5. Can locks ever be de-escalated?**

**Ans:**

No, locks are only escalated, never de-escalated.

**6. What are the different types of lock modes in SQL Server?**

**Ans:**

* Shared
* Update
* Exclusive
* Schema (modification and stability)
* Bulk Update
* Intent (shared, update, exclusive)
* Key Range (shared, insert, exclusive)

**7. Can you explain scenarios where each type of lock would be taken?**

**Ans:**

**SHARED**: Used for read operations that do not change or update data, such as a SELECT statement.

**UPDATE**: Update locks (U) are acquired just prior to modifying the data. If a transaction modifies a row, then the update lock is escalated to an exclusive lock; otherwise, it is converted to a shared lock. Only one transaction can acquire update locks to a resource at one time. Using update locks prevents multiple connections from having a shared lock that want to eventually modify a resource using an exclusive lock. Shared locks are compatible with other shared locks, but are not compatible with Update locks.

**EXCLUSIVE**: Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time.

**INTENT**: Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX). (Another question in the Difficult level section expands on this)

**SCHEMA**: Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S).

**BULK** **UPDATE**: Used when bulk copying data into a table and the TABLOCK hint is specified.

**KEY RANGE**: Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again.

**8. What is lock escalation and what triggers it?**

**Ans:**

The process of converting many fine-grained locks into fewer coarse-grained locks is known as Lock Escalation.

**\*\*** Escalation reduces system resource consumption/overhead while increasing the possibility of concurrency conflicts

**\*\***To escalate locks, the Database Engine attempts to change the intent lock on the table to the corresponding full lock, for example, changing an intent exclusive (IX) lock to an exclusive (X) lock, or an intent shared (IS) lock to a shared (S) lock). If the lock escalation attempt succeeds and the full table lock is acquired, then all heap or B-tree, page (PAGE), key-range (KEY), or row-level (RID) locks held by the transaction on the heap or index are released. If the full lock cannot be acquired, no lock escalation happens at that time and the Database Engine will continue to acquire row, key, or page locks.

**\*\***Lock escalation is triggered at either of these times:

* When a single Transact-SQL statement acquires at least 5,000 locks on a single table or index.
* When the number of locks in an instance of the Database Engine exceeds memory or configuration thresholds.
* If locks cannot be escalated because of lock conflicts, the Database Engine periodically triggers lock escalation at every 1,250 new locks acquired.

**9. Name as many of the lockable resources as possible in SQL Server?**

**Ans:**

* RID (single row on a heap)
* KEY (single row (or range) on an index)
* PAGE
* EXTENT
* HOBT (heap or b-tree)
* TABLE (entire table, all data and indexes)
* FILE
* APPLICATION
* METADATA
* ALLOCATION\_UNIT
* DATABASE

**10. What requirements must be met for a BULK-UPDATE lock to be granted, and what benefit do they server?**

**Ans:**

The Database Engine uses bulk update (BU) locks when bulk copying data into a table, and either the TABLOCK hint is specified or the table lock on bulk load table option is set using sp\_tableoption. Bulk update (BU) locks allow multiple threads to bulk load data concurrently into the same table while preventing other processes that are not bulk loading data from accessing the table.

**11. What is the least restrictive type of lock? What is the most restrictive?**

**Ans:**

The least restrictive type of lock is a shared lock. The most restrictive type of lock is a schema-modification

**12. What is a deadlock and how is it different from a standard block situation?**

**Ans:**

A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock. In a deadlock situation, both transactions in the deadlock will wait forever unless the deadlock is broken by an external process – in a standard blocking scenario, the blocked task will simply wait until the blocking task releases the conflicting lock scenario.

**13. Which 2 isolation levels support optimistic/row-versioned-based concurrency control?**

**Ans:**

First is the READ COMMITTED isolation level. This is the only level that supports both a pessimistic (locking-based) and optimistic (version-based) concurrency control model. Second is SNAPSHOT isolation level that supports only an optimistic concurrency control model.

**14. What database options must be set to allow the use of optimistic models?**

**Ans:**

READ\_COMMITTED\_SNAPSHOT option for the read committed optimistic model. ALLOW\_SNAPSHOT\_ISOLATION option for the snapshot isolation level

**15. What is the size of a lock structure?**

**Ans:**

96 bytes

**16. In what circumstances will you see key-range locks, and what are they meant to protect against?**

**Ans:**

You will only see key-range locks when operating in the SERIALIZABLE isolation level.

* Key-range locks protect a range of rows implicitly included in a record set being read by a Transact-SQL statement. The serializable isolation level requires that any query executed during a transaction must obtain the same set of rows every time it is executed during the transaction. A key range lock protects this requirement by preventing other transactions from inserting new rows whose keys would fall in the range of keys read by the serializable transaction.
* Key-range locking prevents phantom reads. By protecting the ranges of keys between rows, it also prevents phantom insertions into a set of records accessed by a transaction.

**17. Explain the purpose of INTENT locks?**

**Ans:**

The Database Engine uses intent locks to protect placing a shared (S) lock or exclusive (X) lock on a resource lower in the lock hierarchy. Intent locks are named intent locks because they are acquired before a lock at the lower level, and therefore signal intent to place locks at a lower level. Intent locks serve two purposes:

* To prevent other transactions from modifying the higher-level resource in a way that would invalidate the lock at the lower level.
* To improve the efficiency of the Database Engine in detecting lock conflicts at the higher level of granularity.

**18. Can deadlocks occur on resources other than database object?**

**Ans:**

YES.

**19. What are the different types of resources that can deadlock?**

**Ans:**

Deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system, and can occur for resources other than locks on database objects. Here are the resources:

**Locks** – Waiting to acquire locks on resources, such as objects, pages, rows, metadata, and applications can cause deadlock.

**Worker threads** – A queued task waiting for an available worker thread can cause deadlock. If the queued task owns resources that are blocking all worker threads, a deadlock will result

**Memory** – When concurrent requests are waiting for memory grants that cannot be satisfied with the available memory, a deadlock can occur.

**Parallel query execution-related resources** – Coordinator, producer, or consumer threads associated with an exchange port may block each other causing a deadlock usually when including at least one other process that is not a part of the parallel query. Also, when a parallel query starts execution, SQL Server determines the degree of parallelism, or the number of worker threads, based upon the current workload. If the system workload unexpectedly changes, for example, where new queries start running on the server or the system runs out of worker threads, then a deadlock could occur.

**Multiple Active Result Sets (MARS) resources** – Resources used to control interleaving of multiple active requests under MARS, including:

* User resource – when a thread is waiting for a resource that is potentially controlled by a user application, the resource is considered to be an external or user resource and is treated like a lock
* Session mutex – The tasks running in one session are interleaved, meaning that only one task can run under the session at a given time. Before the task can run, it must have exclusive access to the session mutex.
* Transaction mutex – All tasks running in one transaction are interleaved, meaning that only one task can run under the transaction at a given time. Before the task can run, it must have exclusive access to the transaction mutex.

**20. Explain how the database engine manages the memory footprint for the lock pool when running in a dynamic lock management mode.**

**Ans**

**SQL Server 2000:** When the server is started with locks set to 0, the lock manager allocates two percent of the memory allocated to SQL Server to an initial pool of lock structures. As the pool of locks is exhausted, additional locks are allocated. The dynamic lock pool does not allocate more than 40 percent of the memory allocated to SQL Server.

* Generally, if more memory is required for locks than is available in current memory, and more server memory is available (the max server memory threshold has not been reached), SQL Server allocates memory dynamically to satisfy the request for locks. However, if allocating that memory would cause paging at the operating system level (for example, if another application was running on the same computer as an instance of SQL Server and using that memory), more lock space is not allocated.

**SQL Server 2005:** When running in dynamic management mode (i.e. if the server is started with locks configuration option set to 0), the lock manager acquires sufficient memory from the Database Engine for an initial pool of 2,500 lock structures. As the lock pool is exhausted, additional memory is acquired for the pool.

* Generally, if more memory is required for the lock pool than is available in the Database Engine memory pool, and more computer memory is available (the max server memory threshold has not been reached), the Database Engine allocates memory dynamically to satisfy the request for locks. However, if allocating that memory would cause paging at the operating system level (for example, if another application is running on the same computer as an instance of SQL Server and using that memory), more lock space is not allocated. The dynamic lock pool does not acquire more than 60 percent of the memory allocated to the Database Engine. After the lock pool has reached 60 percent of the memory acquired by an instance of the Database Engine, or no more memory is available on the computer, further requests for locks generate an error.

**21. Describe the differences between the pessimistic SERIALIZABLE model and the optimistic SNAPSHOT model in terms of transactional isolation (i.e., not the concurrency differences, but instead how the exact same transactional modifications may result in different final outcomes).**

**Ans:**

* It is typically relatively simple to understand SERIALIZABLE. For the outcome of two transactions to be considered SERIALIZABLE, it must be possible to achieve this outcome by running one transaction at a time in some order.
* Snapshot does not guarantee this level of transactional isolation.
* Imagine the following sample scenario:

**Serializable:**

There is a bag containing a mixture of white and black marbles. Suppose that we want to run two transactions. One transaction turns each of the white marbles into black marbles. The second transaction turns each of the black marbles into white marbles. If we run these transactions under SERIALIZABLE isolation, we must run them one at a time. The first transaction will leave a bag with marbles of only one color. After that, the second transaction will change all of these marbles to the other color. There are only two possible outcomes: a bag with only white marbles or a bag with only black marbles.

**Snapshot:**

If we run these transactions under snapshot isolation, there is a third outcome that is not possible under SERIALIZABLE isolation. Each transaction can simultaneously take a snapshot of the bag of marbles as it exists before we make any changes. Now one transaction finds the white marbles and turns them into black marbles. At the same time, the other transactions finds the black marbles – but only those marbles that where black when we took the snapshot – not those marbles that the first transaction changed to black – and turns them into white marbles. In the end, we still have a mixed bag of marbles with some white and some black. In fact, we have precisely switched each marble.

**22. What are the different isolation levels available?**

**Ans:**

* **Read Uncommitted Isolation Level**
* **Read Committed Isolation Level**
* **Repeatable Read Isolation Level**
* **Serializable Isolation Level**
* **Snapshot Isolation Level**
* **Read Committed Snapshot Isolation Level**

**23. Demonstrate Isolation levels?**

**Ans:**

* **Read Uncommitted**: This is the lowest isolation level. It only isolates transactions and activities to ensure that physically corrupt data is never read. It allows dirty reads, nonrepeatable reads, and phantom reads.
* **Read Committed:** This isolation level does not permit dirty reads, but does allow nonrepeatable reads and phantom reads. This is the default isolation level for SQL Server, and is used for each connection to SQL Server unless one of the other isolation levels has manually been set for a connection.
* **Repeatable Read:** This isolation level does not permit dirty reads or nonrepeatable reads, but does allow phantom reads.
* **Serializable Read:** This is the highest isolation level and ensures that all transactions and statements are completely isolated from each other. It does not allow dirty reads, nonrepeatable reads, or phantom reads.

New isolation levels that introduced in SQL 2005 based on row versioning:

* **READ\_COMMITTED\_SNAPSHOT (statement level):** READ\_COMMITTED\_SNAPSHOT is actually a variation of the default READ\_COMMITTED isolation level. It uses row versioning, instead of locking, to provide read consistency at the SQL Server statement level. When a statement runs that specifies the READ\_COMMITTED isolation level (the default isolation level), and the READ\_COMMITTED\_SNAPSHOT option is turned on at the database level, all statements see a snapshot of the data as it existed at the start of any current transaction. It uses the row-versioned snapshot of the row to return data, and no locking is needed by the statement, which is normally the case. The biggest benefit of this isolation level is that reads do not block writes and writes do not block reads. Writes can still block writes, but this is necessary to prevent data corruption.
* **ALLOW\_SNAPSHOT\_ISOLATION (transaction level):** ALLOW\_SNAPSHOT\_ISOLATION is similar to READ\_COMMITTED\_SNAPSHOT, but it is based at the transaction level, not the statement level. When the ALLOW\_SNAPSHOT\_ISOLATION is turned on at the database level *and* the TRANSACTION ISOLATION LEVEL SNAPSHOT isolation level is turned on for the transaction (using the SET command), all statements see a snapshot of the data as it existed at the start of the transaction.

**24. Any idea about row versioning?**

**Ans:**

The concept of row versioning is not new to SQL Server, as SQL Server has been using it for years with triggers. For example, when a DELETE trigger is executed for a row, a copy of that row is stored in the “deleted table” just in case the trigger is rolled back and the deleted row needs to be “undeleted.” In a sense, the row is versioned, and if need be, can be reused.

Row versioning for isolation levels is very similar, though not identical to row versioning for triggers. When a row versioning-based isolation level (which includes the two new ones we are now discussing) is enabled at the database level, the database engine maintains versions of each row that is modified (for an entire database). Whenever a transaction modifies any row, an image of the row before the modification is copied into a page of what is called the version store. The version store is located in the tempdb database and is used for temporary storage of versioned rows for all of the databases on a single SQL Server instance.

**25. What are the properties of a transaction?**

**Ans:**

There are 4 properties called ACID.

**Atomicity:** All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

**Example**: In an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

**Consistency:** Data is in a consistent state when a transaction starts and when it ends.

**Example:** In an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**Isolation:** The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.

**Example:** in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

**Durability:** After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.

**Example:** in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

**26. How to find out and prevent Deadlocks?**

**Ans:**

**To find Deadlocks**

Error:

Transaction (Process ID 52) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

There are two popular ways to identifying the deadlocks

* **Enabling a Trace Flag**
* By default Deadlocks are not written into sql server errorlog, to do so we have to enable a trace flag.
* Trace Flag 1204 – SQL Server 2000 or below
* Trace Flag 1222 – SQL Server 2005 or above
* Syntax: DBCC TRACEON (1222, -1)

Note: -1 indicates trace should run for all sessions

* **Using the profiler:**
* We need to capture the Lock Events Lock: Deadlock and Lock: Deadlock Chain along with the ObjectID data column

**To prevent Deadlocks:**

* While updating have the application access server objects in the same order each time.
* During transactions, don’t allow any user input. Collect it before the transaction begins.
* Keep transactions as short as possible. To accomplish this when your application does need to read the same data more than once, cache it by storing it in a variable or an array, and then re-reading it from there, not from SQL Server.
* Reduce lock time. Develop your application to grab locks at the latest possible time, and then releases them at the very earliest time.
* If appropriate, reduce lock escalation by using the ROWLOCK or PAGLOCK.
* Consider using the NOLOCK hint to prevent locking if the data being locked is not modified often.
* If appropriate, use low level isolation level according to the possibilities
* Look for other opportunities to improve the efficiency of the queries
* If both deadlock participants are using the same index, consider adding an index that can provide an alternate access path to one of the spids.