In SQL Server, a transaction is a logical unit of a few statements bonded together. A transaction has four properties: atomicity, consistency, isolation, and durability. Atomicity means all the operations performed inside the transaction either commit or roll back together and cannot be partially applied. Consistency means the data modified by committed transactions need to be valid and pass all the constraints and rules. Otherwise, the transaction will roll back the data to the initial state. Isolation means a running transaction will be kept isolated from all other transactions. Durability means committed data resides in the hard drive via the transaction log and will always be available to the end user, even in case of a system failure.

SQL Server uses locks to manage transaction isolation and concurrency. Two basic locks are shared lock (for readers) and exclusive lock (for writers). They can be applied to different ranges, from database to row level.

Isolation affects database concurrency and determines the consistency level of the data the transaction interacts with. There are four isolation levels based on lock system only: *read uncommitted, read committed, repeatable read,* and *serializable*, and two levels based both versioning and lock system: *read\_committed\_snapshot* and *snapshot*.

*Read uncommitted isolation* level results in the most concurrency in the database because it doesn’t require any lock on the resources that are being worked on. Any transaction can manipulate the data, which leads to the most data consistency issues such as dirty reads, lost updates, unrepeatable reads, and phantom reads. *Read committed* is the default isolation level and requires a shared lock on the data. Because shared lock is not compatible with exclusive lock, only other readers (not writers) have access to the held resources, which eliminates dirty reads. However, this isolation level will release the shared lock immediately after read, regardless of the status of the whole transaction. This still gives the writers chances to modify the data. If a transaction happens to update the data between two reads of the same transaction, the two reads can have different outputs. This is known as a non-repeatable read. The other data consistency issues with this isolation level are lost updates and phantom reads. Unlike *read committed*, *repeatable isolation* level requires a shared lock for the reader, which will be held to the end of the transaction; therefore, no writer can modify data. This eliminates non repeatable reads and lost updates. However, *repeatable read isolation* only locks the resources returned by the query from the first run, not those that weren’t there at the first run. In other words, it doesn’t block insert modification, which can lead to phantom reads. The highest isolation level, *serialization*, solves the phantom consistency issue by locking all the possible ranges the query filter could create. No insert modification is allowed at this level. The trade-off is that it results in the least concurrency in the database.

Both *read committed snapshot* and *snapshot isolation* use row versioning. SQL server stores the previous version of committed rows in the *temp* database, which the readers will not be blocked from. In this way, the readers will always receive the last committed data from *temp* database even if the data they are trying to read has an exclusive lock. Although row versioning boosts the concurrency, especially for readers, it can negatively affect manipulation to delete or update the data, because row versioning in the *temp* database needs to be maintained at the same time. Other than row versioning, *read committed snapshot* works similarly to *read committed*, and *snapshot* works like *serializable*.