Q28

Transaction

1. Transaction Introduction

   A transaction is a sequence of operations performed as a single logical unit of work. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).

   The main idea of transactions is that when each of the statements returns an error, the entire modifications rollback to provide data integrity. On the other hand, if all statements are completed successfully the data modifications will become permanent on the database. All these properties are known as the ACID (atomicity, consistency, isolation, durability) in the relational database systems with the first letter of their names.

   1. Atomicity: The entire of the operations that are included by the transaction performed successfully. Otherwise, all operations are canceled at the point of the failure and all the previous operations are rolled back
   2. Consistency: This property ensures that all the data will be consistent after a transaction is completed according to the defined rules, constraints, cascades, and triggers
   3. Isolation: All transactions are isolated from other transactions
   4. Durable: The modification of the committed transactions becomes persist in the database
2. Modes of the Transactions in SQL Server

   SQL Server can operate 3 different transactions modes and these are:

   1. Autocommit Transaction mode is the default transaction for the SQL Server. In this mode, each T-SQL statement is evaluated as a transaction and they are committed or rolled back according to their results. The successful statements are committed and the failed statements are rolled back immediately
   2. Implicit transaction mode enables to SQL Server to start an implicit transaction for every DML statement but we need to use the commit or rolled back commands explicitly at the end of the statements
   3. Explicit transaction mode provides to define a transaction exactly with the starting and ending points of the transaction
3. Save Points in Transactions

   Savepoints can be used to rollback any particular part of the transaction rather than the entire transaction. So that we can only rollback any portion of the transaction where between after the save point and before the rollback command. To define a save point in a transaction we use the SAVE TRANSACTION syntax and then we add a name to the save point.

4. Auto Rollback transactions in SQL Server

   Generally, the transactions include more than one query. In this manner, if one of the SQL statements returns an error all modifications are erased, and the remaining statements are not executed. This process is called Auto Rollback Transaction in SQL.

Locks:

1. Introduction

   Locks are essential to successful SQL Server transactions processing and it is designed to allow SQL Server to work seamlessly in a multi-user environment. Locks are the way that SQL Server manages transaction concurrency.

   SQL Server locks are the essential part of the isolation requirement and it serves to lock the objects affected by a transaction. While objects are locked, SQL Server will prevent other transactions from making any change of data stored in objects affected by the imposed lock. Once the lock is released by committing the changes or by rolling back changes to initial state, other transactions will be allowed to make required data changes.

   Translated into the SQL Server language, this means that when a transaction imposes the lock on an object, all other transactions that require the access to that object will be forced to wait until the lock is released and that wait will be registered with the adequate wait type.

2. Locking has several different aspects:

- Lock duration
- Lock modes
- Lock granularity

  Lock duration specifies a time period during which a resource holds the particular lock. Duration of a lock depends on, among other things, the mode of the lock and the choice of the isolation level.

  Lock modes: shared lock, exclusive lock, update lock:

  1. A *shared lock* reserves a resource (page or row) for reading only. Other processes cannot modify the locked resource while the lock remains. On the other hand, several processes can hold a shared lock for a resource at the same time—that is, several processes can read the resource locked with the shared lock.
  2. An *exclusive lock* reserves a page or row for the exclusive use of a single transaction, , as long as the transaction holds the lock. It is used for DML statements (INSERT, UPDATE, and DELETE) that modify the resource. An exclusive lock cannot be set if some other process holds a shared or exclusive lock on the resource—that is, there can be only one exclusive lock for a resource. Once an exclusive lock is set for the page (or row), no other lock can be placed on the same resource.*The database system automatically chooses the appropriate lock mode according to the operation type (read or write).*
  3. An *update lock* can be placed only if no other update or exclusive lock exists. On the other hand, it can be placed on objects that already have shared locks. (In this case, the update lock acquires another shared lock on the same object.) If a transaction that modifies the object is committed, the update lock is changed to an exclusive lock if there are no other locks on the object. There can be only one update lock for an object.

  Lock Granularity
  Lock granularity specifies which resource is locked by a single lock attempt. The Database Engine can lock the following resources: Row Page Index key or range of index keys Table Extent Database itself.

  A row is the smallest resource that can be locked. The support of row-level locking includes both data rows and index entries. Row-level locking means that only the row that is accessed by an application will be locked. Hence, all other rows that belong to the same page are free and can be used by other applications. The Database Engine can also lock the page on which the row that has to be locked is stored.

Lock granularity affects concurrency. In general, the more granular the lock, the more concurrency is reduced. This means that row-level locking maximizes concurrency because it leaves all but one row on the page unlocked. On the other hand, system overhead is increased because each locked row requires one lock. Page-level locking (and table-level locking) restricts the availability of data but decreases the system overhead.

Lock granularity is essentially the minimum amount of data that is locked as part of a query or update to provide complete isolation and serialization for the transaction. The Lock Manager needs to balance the concurrent access to resources versus the overhead of maintaining a large number of lower-level locks. For example, the smaller the lock size, the greater the number of concurrent users who can access the same table at the same time, but the greater the overhead in maintaining those locks. The greater the lock size, the less overhead that is required to manage the locks, but concurrency is also less. Currently, SQL Server balances performance and concurrency by locking at the row level or higher. Based on a number of factors, such as key distribution, number of rows, row density, search arguments (SARGs), and so on, the query optimizer makes lock granularity decisions internally, and the programmer does not have to worry about such issues.

Isolation Levels:

Transactions specify an isolation level that defines how one transaction is isolated from other transactions. Isolation is the separation of resource or data modifications made by different transactions. Isolation levels are described for which concurrency side effects are allowed, such as dirty reads or phantom reads.

Transaction isolation levels control the following effects:

- Whether locks are taken when data is read, and what type of locks are requested.
- How long the read locks are held.
- Whether read operations referencing rows modified by another transaction:
  - Block until the exclusive lock on the row is freed.
  - Retrieve the committed version of the row that existed at the time the statement or transaction started.
  - Read the uncommitted data modification.

Choosing a transaction isolation level doesn't affect the locks that are acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies. It holds that lock until the transaction completes, whatever the isolation level set for that transaction. For read operations, transaction isolation levels mainly define how the operation is protected from the effects of other transactions.

A lower isolation level increases the ability of many users to access data at the same time. But it increases the number of concurrency effects, such as dirty reads or lost updates, that users might see. Conversely, a higher isolation level reduces the types of concurrency effects that users might see. But it requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level.

The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation. But it uses a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, can retrieve data that has been modified but not committed by other transactions. All concurrency side effects can happen in read uncommitted, however there's no read locking or versioning, so overhead is minimized.

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.
3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

Isolation Level	**Dirty Read**	**Non Repeatable Read**	**Phantom Read**
Read Uncommitted	Yes	Yes	Yes
Read Committed	-	Yes	Yes
Repeatable Read	-	-	Yes
Serializable	-	-	-