1) Difference between temp table and table variable in sql server 2005?

| **Temp Table** | **Table Variable** |
| --- | --- |
| Temp table is valid for a session.  For eg: when you run the following code  create table #temp(i int)  insert into #temp select 345  Go  create table #temp(i int)  insert into #temp select 345  Go  you will get an error | Table variable has a statement-level scope. ie as soon as you execute the statement the scope is lost  For eg: when you run the following code  declare @t table(i int)  insert into @t select 45  GO  declare @t table(i int)  insert into @t select 45  GO  you will not get an error |
| It is possible to alter the temp table to add columns, idexes,etc | It is not possible to alter a table variable |
| It is possible to truncate a temp table | It is not possible to truncate a table variable |
| SELECT INTO method can be used for temp table  SELECT \* INTO #temp from your\_table | SELECT INTO method cannot be used for table variable. You get error for the following  SELECT \* INTO @t from your\_table |
| Temp table can be useful when you have a large amount of data | For small set of data, table variables can be useful |

**Here are a couple of examples:**

| **Description** | **Date Expression** |
| --- | --- |
| Now | SYSDATE |
| Tomorow/ next day | SYSDATE + 1 |
| Seven days from now | SYSDATE + 7 |
| One hour from now | SYSDATE + 1/24 |
| Three hours from now | SYSDATE + 3/24 |
| An half hour from now | SYSDATE + 1/48 |
| 10 minutes from now | SYSDATE + 10/1440 |
| 30 seconds from now | SYSDATE + 30/86400 |
| Tomorrow at 12 midnight | TRUNC(SYSDATE + 1) |
| Tomorrow at 8 AM | TRUNC(SYSDATE + 1) + 8/24 |
| Next Monday at 12:00 noon | NEXT\_DAY(TRUNC(SYSDATE), 'MONDAY') + 12/24 |
| First day of the month at 12 midnight | TRUNC(LAST\_DAY(SYSDATE ) + 1) |
| The next Monday, Wednesday or Friday at 9 a.m | TRUNC(LEAST(NEXT\_DAY(sysdate,''MONDAY' ' ),NEXT\_DAY(sysdate,''WEDNESDAY''), NEXT\_DAY(sysdate,''FRIDAY'' ))) + (9/24) |

#### Table 1 Locking hints available in SQL Server 2000.

| **Locking Hint** | **Description** | **Allowed with** |
| --- | --- | --- |
| HOLDLOCK | Holds the shared locks on the range read, or modified for the duration of the transaction or statement. Overrides the default behavior, which is to release the locks as soon as the data page has been read. HOLDLOCK is equivalent to the SERIALIZABLE transaction isolation level. | SELECT, INSERT, UPDATE, DELETE |
| NOLOCK | Does not honor shared or exclusive locks. NOLOCK is equivalent to the READ UNCOMMITED transaction isolation level. | SELECT |
| PAGLOCK | Forces the transaction to use page-level locks instead of escalating to table-level locks. | SELECT, INSERT, UPDATE, DELETE |
| READCOMMITTED | Equivalent to the READ COMMITTED transaction isolation level, which is the default behavior of SQL Server. | SELECT, INSERT, UPDATE, DELETE |
| READUNCOMMITTED | Same as NOLOCK. Equivalent to the READ UNCOMMITTED transaction isolation level. | SELECT |
| REPEATABLEREAD | Equivalent to the REPEATABLE READ transaction isolation level, which disallows dirty reads, but allows phantoms. | SELECT, INSERT, UPDATE, DELETE |
| ROWLOCK | Forces the transaction to use row-level locking instead of the page- or table-level locking that would otherwise be used. | SELECT, INSERT, UPDATE, DELETE |
| SERIALIZABLE | Equivalent to the SERIALIZABLE transaction isolation level and the HOLDLOCK hint. Holds the shared locks for the duration of the transaction or statement. | SELECT, INSERT, UPDATE, DELETE |
| TABLOCK | Forces SQL Server to use a table-level lock instead of row- or page-level locks. If used with HOLDLOCK, then the lock will be held until the transaction completes. Otherwise, the lock is released as soon as the data is read. For SELECT statements, this hint forces shared table locks. Using this hint with other statements (such as INSERT, UPDATE, or DELETE) will initiate exclusive table lock. | SELECT, INSERT, UPDATE, DELETE |
| TABLOCKX | Places an exclusive lock on the entire table, thereby disabling any other users from reading or writing to the table. | SELECT, INSERT, UDPATE, DELETE |
| UPDLOCK | Places an update lock for the duration of the statement or transaction. This guarantees that data is not changed since it was read. | SELECT, INSERT, UPDATE, DELETE |
| XLOCK | Places an exclusive lock until the end of the transaction. If specified with TABLOCK or PAGLOCK, it will lock the appropriate level. | SELECT |

## Tuning Queries Through Locking Hints

The query hints instruct the SQL Server engine to acquire and release locks in a fashion different from the default behavior. In fact, the term *query hint* is a great understatement. One usually thinks of a "hint" as a friendly suggestion that can be accepted or rejected, and is supposed to be helpful. Query hints, on the other hand, are more of a directive rather than a suggestion. SQL Server has no choice—after I instruct the query optimizer to use a hint, it will adhere to my order as a faithful servant. SQL Server won't debate my suggestion or call me a "bonehead," even if executing the query with my hint is a horrible idea. Therefore, I recommend testing your queries thoroughly with and without using the hints prior to deploying your application in the production environment.

After such a "glowing" introduction, you might think that the locking hints are available just to cause you major headaches. That is not entirely true. Occasionally, you can see a great improvement in your application performance due to using the locking hints, so please don't get discouraged just yet! After all, it never hurts to know and test various alternatives, and go with the best option.

The majority of the locking hints are used with the SELECT statements. A good example of when you might want to use a NOLOCK hint is when you run a report that doesn't have to be 100% accurate. If such a report affects the tables that are being modified, chances are your SELECT statement will have to wait until the other transactions complete. If you use NOLOCK, your query won't have to wait; it will read the uncommitted records. Therefore, your report will be generated much more quickly.

Occasionally, you might find the locking hints useful with the INSERT, UPDATE, and DELETE statements. For instance, if you want to be absolutely sure that no one can modify records while your UDPATE statement is running, you can use the UPDLOCK hint. By default, SQL Server takes shared locks first and then escalates them to UPDLOCK when it's ready to commit an UPDATE transaction. Using UPDLOCK will override such behavior to guarantee that data will be exclusively locked for the duration of the transaction.

Recall from the previous article that locking is a tradeoff between concurrency and the resources used. I've seen an application that used the ROWLOCK hint with every INSERT, UPDATE, and DELETE statement. The idea was to convince SQL Server to lock individual rows each time, thereby reducing locking contingency. Keep in mind that if SQL Server doesn't have enough resources to satisfy your query, it will have to wait until such resources become available. If your query hogs every byte of memory available to SQL Server, then the server will not have enough memory for other tasks. Therefore, consider the pros and cons of using the ROWLOCK hint carefully.

Table 1 lists each available locking hint, discusses its functionality, and tells you which statements support this hint. Keep in mind that using a locking hint that is not supported with a particular statement doesn't generate an error; rather, it has no effect. For instance, you can use NOLOCK with an UPDATE, but SQL Server will ignore that hint—locks will still be acquired. In a similar manner, you can use XLOCK hint with UPDATE, but it won't make any difference.

#### Table 1 Locking hints available in SQL Server 2000.

| **Locking Hint** | **Description** | **Allowed with** |
| --- | --- | --- |
| HOLDLOCK | Holds the shared locks on the range read, or modified for the duration of the transaction or statement. Overrides the default behavior, which is to release the locks as soon as the data page has been read. HOLDLOCK is equivalent to the SERIALIZABLE transaction isolation level. | SELECT, INSERT, UPDATE, DELETE |
| NOLOCK | Does not honor shared or exclusive locks. NOLOCK is equivalent to the READ UNCOMMITED transaction isolation level. | SELECT |
| PAGLOCK | Forces the transaction to use page-level locks instead of escalating to table-level locks. | SELECT, INSERT, UPDATE, DELETE |
| READCOMMITTED | Equivalent to the READ COMMITTED transaction isolation level, which is the default behavior of SQL Server. | SELECT, INSERT, UPDATE, DELETE |
| READUNCOMMITTED | Same as NOLOCK. Equivalent to the READ UNCOMMITTED transaction isolation level. | SELECT |
| REPEATABLEREAD | Equivalent to the REPEATABLE READ transaction isolation level, which disallows dirty reads, but allows phantoms. | SELECT, INSERT, UPDATE, DELETE |
| ROWLOCK | Forces the transaction to use row-level locking instead of the page- or table-level locking that would otherwise be used. | SELECT, INSERT, UPDATE, DELETE |
| SERIALIZABLE | Equivalent to the SERIALIZABLE transaction isolation level and the HOLDLOCK hint. Holds the shared locks for the duration of the transaction or statement. | SELECT, INSERT, UPDATE, DELETE |
| TABLOCK | Forces SQL Server to use a table-level lock instead of row- or page-level locks. If used with HOLDLOCK, then the lock will be held until the transaction completes. Otherwise, the lock is released as soon as the data is read. For SELECT statements, this hint forces shared table locks. Using this hint with other statements (such as INSERT, UPDATE, or DELETE) will initiate exclusive table lock. | SELECT, INSERT, UPDATE, DELETE |
| TABLOCKX | Places an exclusive lock on the entire table, thereby disabling any other users from reading or writing to the table. | SELECT, INSERT, UDPATE, DELETE |
| UPDLOCK | Places an update lock for the duration of the statement or transaction. This guarantees that data is not changed since it was read. | SELECT, INSERT, UPDATE, DELETE |
| XLOCK | Places an exclusive lock until the end of the transaction. If specified with TABLOCK or PAGLOCK, it will lock the appropriate level. | SELECT |

The syntax for using the locking hints is very simple. You add the WITH keyword to the FROM clause, and include a locking hint in the parentheses that follow. For example, the following statement returns a few columns from the sales table in the pubs database without honoring any shared or exclusive locks.

SELECT stor\_id, ord\_num, qty FROM sales WITH (NOLOCK) WHERE payterms = 'Net 30'

The following statement gives all employees who started before the first of the year a 20% raise, and forces SQL Server to use the page-level locks instead of escalating to a single table-level lock:

UPDATE employee SET salary = salary \* 1.20 FROM employee WITH (PAGLOCK)

WHERE start\_date < '1/1/2002'

Please note that the locking hints and the transaction isolation levels might conflict with each other. When this happens, SQL Server honors the more restrictive of the two. For instance, the following query holds the locks on the employee table, even though the isolation level instructs SQL Server to ignore the locks:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED   
GO  
BEGIN TRANSACTION  
 SELECT \* FROM employee WITH (HOLDLOCK)  
COMMIT TRANSACTION

Also, note that you can specify multiple locking hints in the same query. For instance, the following group of statements instructs SQL Server to hold a table-level lock for the duration of the transaction:

BEGIN TRAN  
 SELECT fname AS first\_name, minit AS middle\_initial, lname AS last\_name FROM employee WITH (TABLOCK, HOLDLOCK) COMMIT

The conflicting locking hints generates an error, as shown in the following statement:

SELECT \* FROM Employee WITH (HOLDLOCK, NOLOCK)   
Result:  
Server: Msg 1047, Level 15, State 1, Line 10  
Conflicting locking hints specified.

## Changing Locking Behavior with SET Options

In addition to SET TRANSACTION ISOLATION LEVEL, there are several other SET options that let you customize SQL Server's locking behavior. Set options affect only a single connection; they don't alter locking on any other connections. It is usually a good idea to execute SET statements at the beginning of the session so you don't get two behaviors on the same connection. As soon as the user disconnects all the SET options go out of scope.

Table 2 lists the SET options affecting the default locking behavior, and describes each option:

#### Table 2 SET options affecting SQL Server locking behavior.

| **SET Option** | **Description** |
| --- | --- |
| SET IMPLICIT\_TRANSACTONS ON (OFF) | This setting defaults to OFF, and determines whether certain statements should be explicitly committed or rolled back. This setting affects how SQL Server treats the following statements:  ALTER TABLE  TRUNCATE TABLE  CREATE (any object)  DELETE  DROP (any object)  FETCH  GRANT  INSERT  OPEN  REVOKE  SELECT  UPDATE  Note that if the statement happens within a transaction, it does not start a new transaction. In addition, this setting has no effect on the transactions that are open when this statement is executed. |
| SET XACT\_ABORT ON (OFF) | When this option is turned on, it automatically rolls back all transactions in a batch if any statement within the batch encounters an error.  If this option is turned off (default), then only the statements within the transaction that encountered the error are rolled back while all other commands are allowed to complete. |
| SET CURSOR\_CLOSE\_ON\_COMMIT ON (OFF) | This setting closes all open cursors when a transaction containing the cursor is committed or rolled back. If the setting is OFF (default), then cursors remain open and must be explicitly closed.  Make a note, though, that it's a good coding practice to explicitly close and deallocate your cursors, regardless of the setting. |
| SET DEADLOCK\_PRIORITY LOW (NORMAL) | The LOW setting advises SQL Server that the current connection should have a "better" chance of being chosen as a victim if a deadlock occurs. Otherwise, it's up to SQL Server to determine which of the connections involved in the deadlock is the "lucky" victim. The default priority is NORMAL. |
| SET LOCK\_TIMEOUT timeout\_period | This setting specifies the number of milliseconds to wait until the error is returned if the requested lock cannot be acquired. By default, SQL Server will try indefinitely. The default value of this setting is –1. The value of zero will instruct SQL Server to return an error as soon as the lock is encountered. |
| SET REMOTE\_PROC\_TRANSACTIONS ON (OFF) | Occasionally, you might have a transaction that spans multiple servers. The distributed transactions are managed by the Distributed Transaction Coordinator (MS DTC) by using a two-step process referred to as two-phase commit (2PC). Such transactions are supported only if MS DTC is running on all participating servers. Distributed transactions are difficult to debug because they go across the network and execute on multiple servers. We suggest using them sparingly.  The SET REMOTE\_PROC\_TRANSACTIONS setting starts a distributed transaction when a remote stored procedure is executed within a local transaction. The default is OFF. |