<http://blog.csdn.net/huwei2003/article/details/4047191>

# sql server锁知识及锁应用

转载 2017年04月01日 18:30:20

* 标签：
* [sql server](http://so.csdn.net/so/search/s.do?q=sql%20server&t=blog) /
* [数据库](http://so.csdn.net/so/search/s.do?q=数据库&t=blog) /
* [lock](http://so.csdn.net/so/search/s.do?q=lock&t=blog) /
* [nolock](http://so.csdn.net/so/search/s.do?q=nolock&t=blog)

# sql server锁（lock）知识及锁应用

提示：这里所摘抄的关于锁的知识有的是不同sql server版本的，对应于特定版本时会有问题。

## 一 关于锁的基础知识

### (一). 为什么要引入锁

当多个用户同时对数据库的并发操作时会带来以下数据不一致的问题:

**◆丢失更新**

A,B两个用户读同一数据并进行修改,其中一个用户的修改结果破坏了另一个修改的结果,比如订票系统

**◆脏读**

A用户修改了数据,随后B用户又读出该数据,但A用户因为某些原因取消了对数据的修改,数据恢复原值,此时B得到的数据就与数据库内的数据产生了不一致

**◆不可重复读**

A用户读取数据,随后B用户读出该数据并修改,此时A用户再读取数据时发现前后两次的值不一致

并发控制的主要方法是封锁,锁就是在一段时间内禁止用户做某些操作以避免产生数据不一致

### (二) 锁的分类

◆锁的类别有两种分法：

**1. 从数据库系统的角度来看：**分为独占锁（即排它锁），共享锁和更新锁

MS-SQL Server 使用以下资源锁模式。

锁模式 描述

共享 (S) 用于不更改或不更新数据的操作（只读操作），如 SELECT 语句。

更新 (U) 用于可更新的资源中。防止当多个会话在读取、锁定以及随后可能进行的资源更新时发生常见形式的死锁。

排它 (X) 用于数据修改操作，例如 INSERT、UPDATE 或 DELETE。确保不会同时同一资源进行多重更新。

意向锁 用于建立锁的层次结构。意向锁的类型为：意向共享 (IS)、意向排它 (IX) 以及与意向排它共享 (SIX)。

架构锁 在执行依赖于表架构的操作时使用。架构锁的类型为：架构修改 (Sch-M) 和架构稳定性 (Sch-S)。

大容量更新 (BU) 向表中大容量复制数据并指定了 TABLOCK 提示时使用。

**◆共享锁**

共享 (S) 锁允许并发事务读取 (SELECT) 一个资源。资源上存在共享 (S) 锁时，任何其它事务都不能修改数据。一旦已经读取数据，便立即释放资源上的共享 (S) 锁，除非将事务隔离级别设置为可重复读或更高级别，或者在事务生存周期内用锁定提示保留共享 (S) 锁。

**◆更新锁**

更新 (U) 锁可以防止通常形式的死锁。一般更新模式由一个事务组成，此事务读取记录，获取资源（页或行）的共享 (S) 锁，然后修改行，此操作要求锁转换为排它 (X) 锁。如果两个事务获得了资源上的共享模式锁，然后试图同时更新数据，则一个事务尝试将锁转换为排它 (X) 锁。共享模式到排它锁的转换必须等待一段时间，因为一个事务的排它锁与其它事务的共享模式锁不兼容；发生锁等待。第二个事务试图获取排它 (X) 锁以进行更新。由于两个事务都要转换为排它 (X) 锁，并且每个事务都等待另一个事务释放共享模式锁，因此发生死锁。

若要避免这种潜在的死锁问题，请使用更新 (U) 锁。一次只有一个事务可以获得资源的更新 (U) 锁。如果事务修改资源，则更新 (U) 锁转换为排它 (X) 锁。否则，锁转换为共享锁。

**◆排它锁**

排它 (X) 锁可以防止并发事务对资源进行访问。其它事务不能读取或修改排它 (X) 锁锁定的数据

**◆意向锁**

意向锁表示 SQL Server 需要在层次结构中的某些底层资源上获取共享 (S) 锁或排它 (X) 锁。例如，放置在表级的共享意向锁表示事务打算在表中的页或行上放置共享 (S) 锁。在表级设置意向锁可防止另一个事务随后在包含那一页的表上获取排它 (X) 锁。意向锁可以提高性能，因为 SQL Server 仅在表级检查意向锁来确定事务是否可以安全地获取该表上的锁。而无须检查表中的每行或每页上的锁以确定事务是否可以锁定整个表。

意向锁包括意向共享 (IS)、意向排它 (IX) 以及与意向排它共享 (SIX)。

锁模式 描述

意向共享 (IS) 通过在各资源上放置 S 锁，表明事务的意向是读取层次结构中的部分（而不是全部）底层资源。

意向排它 (IX) 通过在各资源上放置 X 锁，表明事务的意向是修改层次结构中的部分（而不是全部）底层资源。IX 是 IS 的超集。

与意向排它共享 (SIX) 通过在各资源上放置 IX 锁，表明事务的意向是读取层次结构中的全部底层资源并修改部分（而不是全部）底层资源。允许顶层资源上的并发 IS 锁。例如，表的 SIX 锁在表上放置一个 SIX 锁（允许并发 IS 锁），在当前所修改页上放置 IX 锁（在已修改行上放置 X 锁）。虽然每个资源在一段时间内只能有一个 SIX 锁，以防止其它事务对资源进行更新，但是其它事务可以通过获取表级的 IS 锁来读取层次结构中的底层资源。

**◆独占锁：**

只允许进行锁定操作的程序使用，其他任何对他的操作均不会被接受。执行数据更新命令时，SQL Server会自动使用独占锁。当对象上有其他锁存在时，无法对其加独占锁。

共享锁：共享锁锁定的资源可以被其他用户读取，但其他用户无法修改它，在执行Select时，SQL Server会对对象加共享锁。

**◆更新锁：**

当SQL Server准备更新数据时，它首先对数据对象作更新锁锁定，这样数据将不能被修改，但可以读取。等到SQL Server确定要进行更新数据操作时，他会自动将更新锁换为独占锁，当对象上有其他锁存在时，无法对其加更新锁。

**2. 从程序员的角度看：分为乐观锁和悲观锁。**

**◆乐观锁：**完全依靠数据库来管理锁的工作。

**◆悲观锁：**程序员自己管理数据或对象上的锁处理。

MS-SQLSERVER 使用锁在多个同时在数据库内执行修改的用户间实现悲观并发控制

### 三 锁的粒度

锁粒度是被封锁目标的大小,封锁粒度小则并发性高,但开销大,封锁粒度大则并发性低但开销小

SQL Server支持的锁粒度可以分为为行、页、键、键范围、索引、表或数据库获取锁

资源 描述

RID 行标识符。用于单独锁定表中的一行。

键 索引中的行锁。用于保护可串行事务中的键范围。

页 8 千字节 (KB) 的数据页或索引页。

扩展盘区 相邻的八个数据页或索引页构成的一组。

表 包括所有数据和索引在内的整个表。

DB 数据库。

SQL Server 提供以下的锁级别:  
  
**DATABASE**  -- 无论何时当一个SQL Server 进程正在使用除master以外的数据库时，Lock Manager为该进程授予数据库级的锁。数据库级的锁总是共享锁，用于跟踪何时数据库在使用中，以防其他进程删除该数据库，将数据库置为脱机，或者恢复数据库。注意，由于master和tempdb数据库不能被删除或置为脱机，所以不需要在它们之上加锁。  
**FILE** -- 文件级的锁用于锁定数据库文件。  
**EXTENT** -- Extent锁用于锁定extents,通常仅在空间分配和重新分配的时候使用。一个extent由8个连续的数据页或索引页组成。Extent锁可以是共享锁也可以是独占锁。  
**ALLOCATION\_UNIT** -- 使用在数据库分配单元上。  
**TABLE** -- 这种级别的锁将锁定整个表，包括数据和索引。何时将获得表级锁的例子包括在Serializable隔离级别下从包含大量数据的表中选取所有的行，以及在表上执行不带过滤条件的update或delete。  
**Heap or B-Tree (HOBT)** -- 用于堆数据页，或者索引的二叉树结构。  
**PAGE** -- 使用页级锁，由8KB数据或者索引信息组成的整个页被锁定。当需要读取一页的所有行或者需要执行页级别的维护如页拆分后更新页指针时，将会获取页级锁。  
**Row ID (RID**) -- 使用RID锁，页内的单一行被锁定。无论何时当提供最大化的资源并发性访问是有效并且可能时，将获得RID锁。  
**KEY** -- SQL Server使用两种类型的Key锁。其中一个的使用取决于当前会话的锁隔离级别。对于运行于Read Committed 或者 Repeatable Read 隔离模式下的事务，SQL Server 锁定与被访问的行相关联的的实际索引key。（如果是表的聚集索引，数据行位于索引的叶级。行上在这些你看到的是Key锁而不是行级锁。）若在Serializable隔离模式下，通过锁定一定范围的key值从而不允许新的行插入到该范围内，SQL Server防止了“幻读”。这些锁因而被称作“key-range lock”。  
**METADATA** -- 用于锁定系统目录信息（元数据）。  
**APPLICATION** -- 允许用户定义他们自己的锁，指定资源名称、锁模式、所有者、timeout间隔。

### ****四 SQL Server 锁类型(与粒度相对应)****

1． HOLDLOCK: 在该表上保持共享锁，直到整个事务结束，而不是在语句执行完立即释放所添加的锁。　　   
2． NOLOCK：不添加共享锁和排它锁，当这个选项生效后，可能读到未提交读的数据或“脏数据”，这个选项仅仅应用于SELECT语句。　　   
3． PAGLOCK：指定添加页锁（否则通常可能添加表锁）。　   
4． READCOMMITTED用与运行在提交读隔离级别的事务相同的锁语义执行扫描。默认情况下，SQL Server 2000 在此隔离级别上操作。   
5． READPAST: 跳过已经加锁的数据行，这个选项将使事务读取数据时跳过那些已经被其他事务锁定的数据行，而不是阻塞直到其他事务释放锁，READPAST仅仅应用于READ COMMITTED隔离性级别下事务操作中的SELECT语句操作。　　  
6． READUNCOMMITTED：等同于NOLOCK。　　   
7． REPEATABLEREAD：设置事务为可重复读隔离性级别。　   
8． ROWLOCK：使用行级锁，而不使用粒度更粗的页级锁和表级锁。　　   
9． SERIALIZABLE：用与运行在可串行读隔离级别的事务相同的锁语义执行扫描。等同于 HOLDLOCK。　   
10． TABLOCK：指定使用表级锁，而不是使用行级或页面级的锁，SQL Server在该语句执行完后释放这个锁，而如果同时指定了HOLDLOCK，该锁一直保持到这个事务结束。　　  
11． TABLOCKX：指定在表上使用排它锁，这个锁可以阻止其他事务读或更新这个表的数据，直到这个语句或整个事务结束。　   
12． UPDLOCK ：指定在读表中数据时设置更新 锁（update lock）而不是设置共享锁，该锁一直保持到这个语句或整个事务结束，使用UPDLOCK的作用是允许用户先读取数据（而且不阻塞其他用户读数据），并且保证在后来再更新数据时，这一段时间内这些数据没有被其他用户修改。

### 五 锁定时间的长短

锁保持的时间长度为保护所请求级别上的资源所需的时间长度。

用于保护读取操作的共享锁的保持时间取决于事务隔离级别。采用 READ COMMITTED 的默认事务隔离级别时，只在读取页的期间内控制共享锁。在扫描中，直到在扫描内的下一页上获取锁时才释放锁。如果指定 HOLDLOCK 提示或者将事务隔离级别设置为 REPEATABLE READ 或 SERIALIZABLE，则直到事务结束才释放锁。

根据为游标设置的并发选项，游标可以获取共享模式的滚动锁以保护提取。当需要滚动锁时，直到下一次提取或关闭游标（以先发生者为准）时才释放滚动锁。但是，如果指定 HOLDLOCK，则直到事务结束才释放滚动锁。

用于保护更新的排它锁将直到事务结束才释放。

如果一个连接试图获取一个锁，而该锁与另一个连接所控制的锁冲突，则试图获取锁的连接将一直阻塞到：

将冲突锁释放而且连接获取了所请求的锁。

连接的超时间隔已到期。默认情况下没有超时间隔，但是一些应用程序设置超时间隔以防止无限期等待

### 六 SQL Server 中锁的自定义

◆处理死锁和设置死锁优先级

死锁就是多个用户申请不同封锁,由于申请者均拥有一部分封锁权而又等待其他用户拥有的部分封锁而引起的无休止的等待

可以使用SET DEADLOCK\_PRIORITY控制在发生死锁情况时会话的反应方式。如果两个进程都锁定数据，并且直到其它进程释放自己的锁时，每个进程才能释放自己的锁，即发生死锁情况。

◆2 处理超时和设置锁超时持续时间。

@@LOCK\_TIMEOUT 返回当前会话的当前锁超时设置，单位为毫秒

SET LOCK\_TIMEOUT 设置允许应用程序设置语句等待阻塞资源的最长时间。当语句等待的时间大于 LOCK\_TIMEOUT 设置时，系统将自动取消阻塞的语句，并给应用程序返回"已超过了锁请求超时时段"的 1222 号错误信息

示例

下例将锁超时期限设置为 1,800 毫秒。

SET LOCK\_TIMEOUT 1800

◆设置事务隔离级别。

◆对 SELECT、INSERT、UPDATE 和 DELETE 语句使用表级锁定提示。

◆配置索引的锁定粒度

可以使用 sp\_indexoption 系统存储过程来设置用于索引的锁定粒度

### 七 查看锁的信息

1 执行 EXEC SP\_LOCK 报告有关锁的信息

2 查询分析器中按Ctrl+2可以看到锁的信息

### 八 使用注意事项

**如何避免死锁，最小化锁竞争**

1 使用事务时，尽量缩短事务的逻辑处理过程，及早提交或回滚事务，事务持有锁的时间越短，锁竞争发生的机会就越少；将不是事务所管理的工作单元锁必需的命令移出事务。；

2 设置死锁超时参数为合理范围，如：3分钟-10分种；超过时间，自动放弃本次操作，避免进程悬挂；

3 优化程序，检查并避免死锁现象出现；

4 .对所有的脚本和SP都要仔细测试，在正是版本之前。

5 所有的SP都要有错误处理（通过@error）

6 一般不要修改SQL SERVER事务的默认级别。不推荐强行加锁

7 将组成事务的语句作为一个的单独的批命令处理，以消除 BEGIN TRAN 和 COMMIT  TRAN 语句之间的网络延迟造成的不必要的延迟。

8 考虑完全地使用存储过程编写事务代码。典型地，存储过程比批命令运行更快。

9 在游标中尽可早地Commit更新。因为游标处理比面向集合的处理慢得多，因此导致锁被持有的时间更久。  
  
10 使用每个进程所需的最低级别的锁隔离。比如说，如果脏读是可接受的并且不要求结果必须精确，那么可以考虑使用事务隔离级别0（Read Uncommitted）,仅在绝对必要时才使用Repeatable Read or Serializable隔离级别。  
  
11 在 BEGIN TRAN 和 COMMIT TRAN 语句之间，绝不允许用户交互，因为这样做可能锁被持有无限期的时间。

### 九 几个有关锁的问题

**1 如何锁一个表的某一行**

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT \* FROM table ROWLOCK WHERE id = 1

**2 锁定数据库的一个表**

SELECT \* FROM table WITH (HOLDLOCK)

加锁语句：

sybase:   
update 表 set col1=col1 where 1=0 ;   
MSSQL:   
select col1 from 表 (tablockx) where 1=0 ;   
oracle:   
LOCK TABLE 表 IN EXCLUSIVE MODE ；   
加锁后其它人不可操作，直到加锁用户解锁，用commit或rollback解锁

◆排它锁

新建两个连接,在第一个连接中执行以下语句

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. begin tran
2. update table1 set A='aa' where B='b2'
3. waitfor delay '00:00:30' --等待30秒
4. commit tran
5. --在第二个连接中执行以下语句
6. begin tran
7. select \* from table1 where B='b2'
8. commit tran

若同时执行上述两个语句，则select查询必须等待update执行完毕才能执行即要等待30秒

◆共享锁

在第一个连接中执行以下语句

**[html]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. begin tran
2. select \* from table1 holdlock  --holdlock人为加锁
3. where B='b2'
4. waitfor delay '00:00:30' --等待30秒
5. commit tran

◆共享锁

在第一个连接中执行以下语句

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. begin tran
2. select \* from table1 holdlock --holdlock人为加锁
3. where B='b2'
4. waitfor delay '00:00:30' --等待30秒
5. commit tran

在第二个连接中执行以下语句

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. begin tran
2. select A,C from table1 where B='b2'
3. update table1 set A='aa' where B='b2'
4. commit tran

若同时执行上述两个语句，则第二个连接中的select查询可以执行

而update必须等待第一个事务释放共享锁转为排它锁后才能执行 即要等待30秒

◆死锁

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. --在第一个连接中执行以下语句
2. begin tran
3. update table1  set A='aa' where B='b2'
4. waitfor delay '00:00:30'
5. update table2  set D='d5'  where E='e1'
6. commit tran
8. --在第二个连接中执行以下语句
10. begin tran
11. update table2  set D='d5'  where E='e1'
12. waitfor delay '00:00:10'
13. update table1  set A='aa'  where B='b2'
14. commit tran

同时执行，系统会检测出死锁，并中止进程

### 十 应用程序锁:

应用程序锁就是客户端代码生成的锁，而不是sql server本身生成的锁

处理应用程序锁的两个过程

sp\_getapplock 锁定应用程序资源

sp\_releaseapplock 为应用程序资源解锁

注意: 锁定数据库的一个表的区别

SELECT \* FROM table WITH (HOLDLOCK) 其他事务可以读取表，但不能更新删除

SELECT \* FROM table WITH (TABLOCKX) 其他事务不能读取表,更新和删除

交读事务使用行版本控制。   
使用快照隔离。  
使用绑定连接。

## 二 锁的分析及应用系列

### 1 用SqlServer Profile来查看分析锁的信息

　　这个工具我想大家都明白，它的监视能力真的是无所不能。。。锁的痉挛状态也全在它的掌握之中。

1. 首先我做一个Person表，Name字段设定4000字节，这样一个数据页可以容纳2条数据，如下图：

DROP TABLE dbo.Person

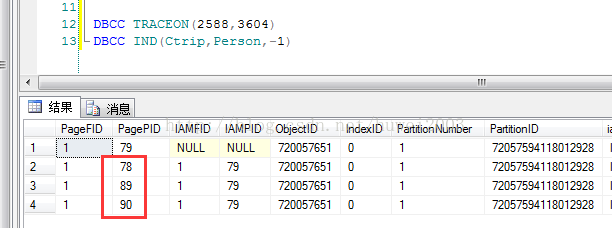
CREATE TABLE Person(ID INT IDENTITY,NAME CHAR(**4000**) DEFAULT 'aaaaa')

--插入6条，生成3个数据页

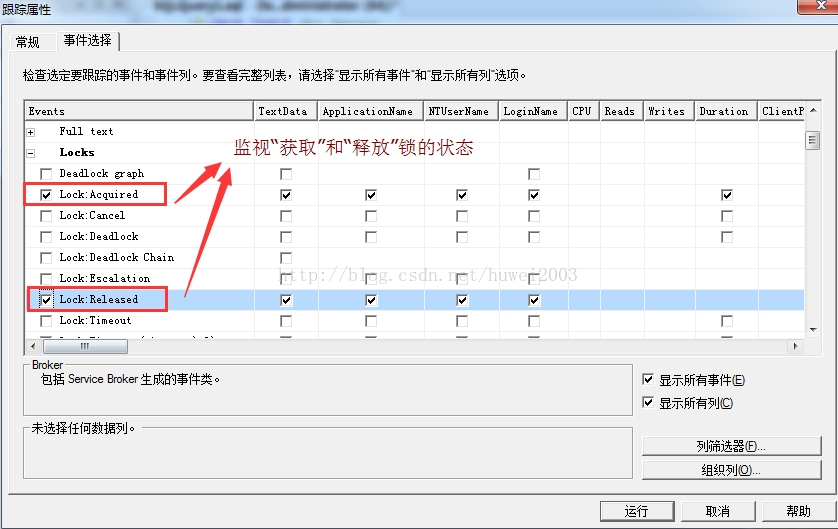
INSERT INTO dbo.Person DEFAULT VALUES

go **6**

2. 下面我们看看数据在数据页的分布情况。

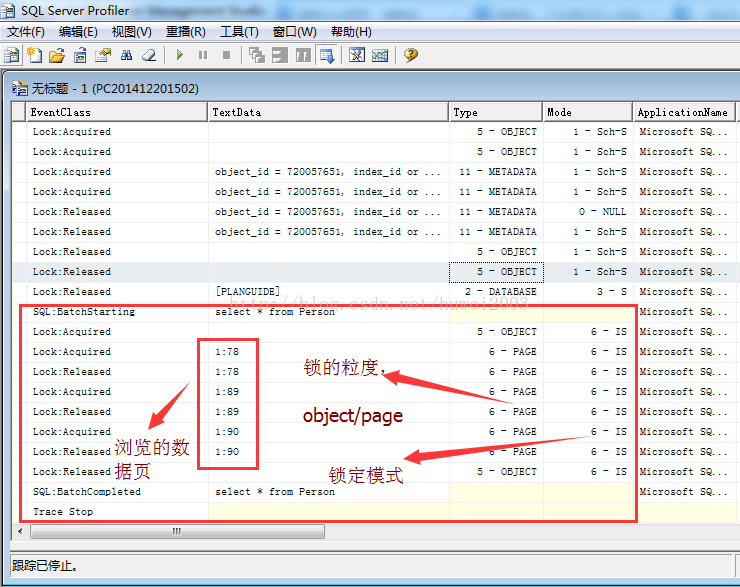


3. 然后我们开启Profile，在“事件选择”的Events中选择”Lock:Acquired“和”Lock:Released“ ，然后运行，如下图：



使用测试数据

1. 首先我执行一个简单的 SELECT \* FROM dbo.Person，看看表/数据页/记录的加锁情况。



从图中可以看到，select执行的大概步骤如下：

第一步：给表（Object）加上IS（意向共享锁）。

第二步：先给1:78号数据页加IS锁，扫描78号数据页，然后释放IS锁。

第三步：同样的道理扫描之后的数据页。

第四步：最后释放表的IS锁，结束整个锁流程。

看完上面的一系列的Lock:Acquired 和 Lock:Released的话，你有没有发现一个问题，不是说好给记录（RID）加上S锁么？？？这里没加，是因为引擎进入78号数据页的时候，未发现它存在IU锁或者IX锁。。。所以。。。这个属于锁的组合，后续会说。

2. 接下来用UPDATE dbo.Person SET NAME='bbbbb' WHERE ID=3来看看update的整个过程，乍一看，Profile捕获到的记录还是比较多的，下面具体看图：



 第一步： 给表(Object)加上IX锁，

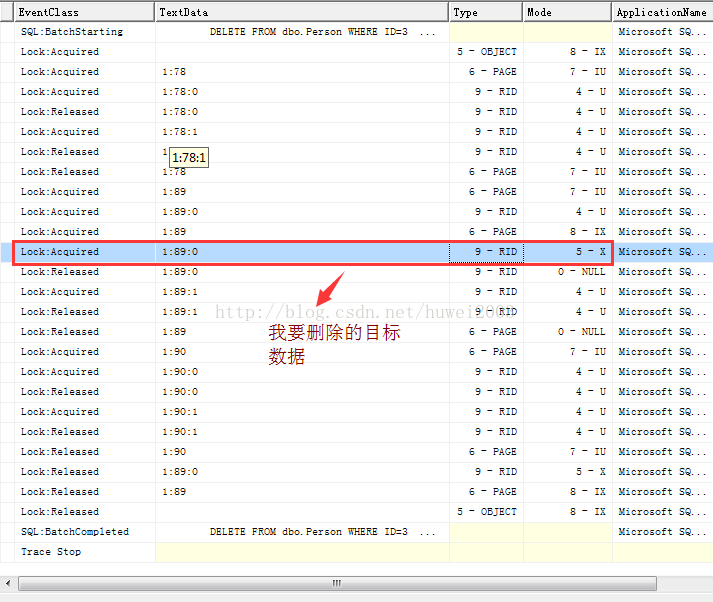
 第二步： 给数据页(1:78)数据页分配IU锁。然后开始逐一扫描78号数据页的RID记录，进入前就Acquired，退出后就Released，当扫描完78号数据页的所有RID后，再释放78                  号数据页的IU锁，进入下一个数据页。。。

 第三步： 我们发现ID=3是在89号数据页上，当引擎扫到该RID之后，我们观察到89号的数据页由IU锁变成了IX锁，并且把1:89:0（slot为0的记录）由U锁变成X锁，变成X锁                       后，就排斥了其他所有的锁，这时候就可以进行Update操作了。

 第四步： 后面就继续90号数据页，步骤类似，第二步和第三步。

不知道细心的你有没有发现，在Released Object之前我们才释放1:89:0的X锁，然后释放89号数据页的IX锁，这说明什么？？？说明这个Update是贯穿于这个事务的，不像Select操作中，扫完一个数据页就释放一个数据页。

3. 最后再看一个DELETE FROM dbo.Person WHERE ID=3 的操作。



　　大概扫了一下上面的图，或许你感觉和Update操作大差不差，会扫描数据页中的每个记录并加上U锁。当在1:89:0槽位中找到了目标记录后，然后将U锁转化为X锁，具体可以参考Update。

### 2 深入的探讨锁机制

　　上一篇我只是做了一个堆表让大家初步的认识到锁的痉挛状态，但是在现实世界上并没有这么简单的事情，起码我的表不会没有索引对吧，，，还有就是我的表一定会有很多的连接过来，10:1的读写，很多码农可能都会遇到类似神乎其神的死锁，卡住，读不出来，插不进入等等神仙的事情导致性能低下，这篇我们一起来探讨下。

一： 当select遇到性能低下的update会怎么样？

1. 还是使用原始的person表，插入6条数据，由于是4000字节，所以两条数据就是一个数据页，如下图：

1 DROP TABLE dbo.Person

2 CREATE TABLE Person(ID INT IDENTITY,NAME CHAR(**4000**) DEFAULT 'aaaaa')

3 --插入6条数据，刚好3个数据页

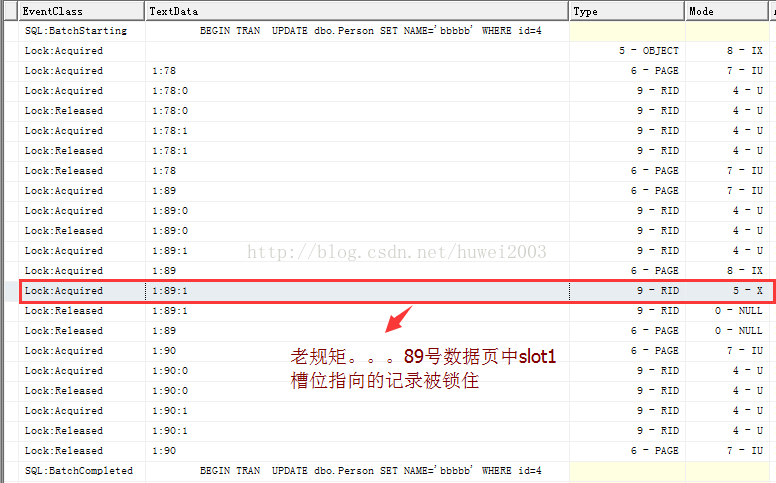
4 INSERT INTO dbo.Person DEFAULT VALUES

5 go **6**

2. 为了模拟性能低下的Update操作，我们开个显式事务来更新ID=4的记录，并且用profile看一下，如下图：

1 BEGIN TRAN

2 UPDATE dbo.Person SET NAME='bbbbb' WHERE id=**4**



3. 然后我们开下另一个会话连接，读取ID=6的记录会是怎样？好奇吗？

1 SELECT \* FROM Person WHERE ID=**6**



从上面流程你是否看到，当扫描到89号数据页的slot1槽位的时候卡住了。。。我想你应该知道update正好已经给这条记录加上了X锁。。。如果你够细心，你还会发现，给S锁附加记录的条件是在当引擎发现记录所在的数据页已经附加上了IX锁的情况下，才给该号数据页下的每条记录附加S锁，对吧。。。好了，既然在Profile上面看不到了，我还是有其他办法来判断到底select语句现在处于什么状态。

4. 使用sys.dm\_tran\_locks来看当前各个连接持有锁的状态。

复制代码

1 SELECT l.request\_session\_id,

2 DB\_NAME(l.resource\_database\_id),OBJECT\_NAME(p.object\_id),

3 l.resource\_description,l.request\_type,

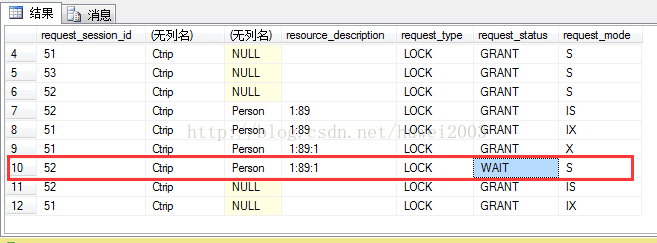
4 l.request\_status,request\_mode

5 FROM sys.dm\_tran\_locks AS l

6 LEFT JOIN sys.partitions AS p

7 ON l.resource\_associated\_entity\_id=p.hobt\_id

复制代码



仔细观察上图可以看到，当前有51和52号会话，51号在1:89:1槽位上使用了X锁并且没有释放，52号此时也进入了1:89:1中，并且想给该RowID附加S锁，但是你也知道S和X锁是排斥的，所以很无奈的一直保持等待状态。

二：使用索引或许可以帮你逃过一劫

　　当你看完上面的讲叙，是不是有点害怕？？？要是在生产环境下出现了这种情况，那我们是不是死的很惨？？？那接下来使用索引是不是真的可以帮我们躲过一劫呢？下面跟我一起看一看。

1. 新建索引index

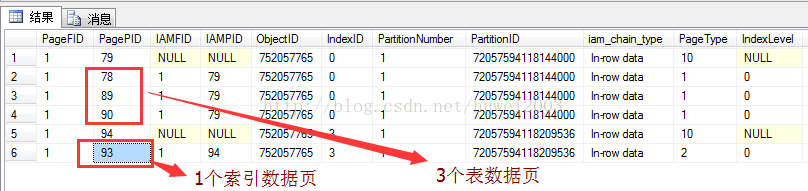
1 -- 在ID列上建一个index

2 CREATE INDEX idx\_person ON dbo.Person(ID)

2. 然后我们看下数据页的分布情况，可以看到下图中78,89,90是表数据页，93号为索引数据页。

1 DBCC TRACEON(**2588**,**3604**)

2 DBCC IND(Ctrip,Person,-**1**)



3. 麻蛋的，继续执行上面的那个慢update

BEGIN TRAN

UPDATE dbo.Person SET NAME='bbbbb' WHERE id=**4**

4. 激动人心的时刻来了，由于数据太少，所以我这里强制让引擎执行我创建的索引，看看结果怎样？



居然没卡住？？？现在是不是有一股强烈的好奇心来了，狗狗狗。。。马上开启profile，看看到底都发生了什么？



仔细看完这个图，是不是觉得很有意思呢？？？具体步骤如下：

第一步：给表（Object）加上IS锁。

第二步：因为要走索引，给93号索引数据页加上IS锁。

第三步：找到93号索引数据页的目标key，给这个key加上S锁，有人可能就会问了。。。这个key不就是6嘛，为什么这个key=(61005a25560e)，你要是太好奇我可以告                   诉你，年轻人说话不要太屌，每行索引记录都有一个散列值，这个值就是根据索引的几个字段散列出来的，好处就是防止你的索引长度过大，导致锁这个记录的                   时候太耗费锁空间了。。。。如果你还是不太相信的话，我用DBCC给你看一看。



第四步：根据这个key直接跳到存放记录的90号数据页中，万幸的是update的记录刚好不在90号数据页中。。。。就这样躲过一劫了。。。然后select顺利的读取到了该                  读的记录，最后释放相关的IS锁。

### 3 nolock引发的三级事件的一些思考

　　曾今有件事情让我记忆犹新，那年刚来携程不久，马上就被安排写一个接口，供企鹅公司调用他们员工的差旅信息，然后我就三下五除二的给写好了，上线之后，大概过了一个月。。。DBA那边报告数据库出现大量锁超时，并且及时根据sql的来源将email发到了我们部门，指出sql读取时间过长，并且缺少nolock，影响了大量机票订单入库，然后我就拿着sql去生产环境跑了下，22s。。。花擦。。。项目上线时间太久，版本已经不存在了，无法回滚。。。原本准备撤下接口。。。看了下撤下接口跟加上nolock时间相差不多，最后决定先加上nolock，发布紧急单。。。然后再优化，DBA那边暂时做手工解锁，发上去后，最后就是损失XXXX订单。。。定级为三级事件。然后就是追责，当然这个责任只能有老大们去承担了，出了这次由我引发的事件，我得思考了，出了事情对我不见得全是坏事，起码这次会让我铭记如心，想想也搓，来携程之前根本就不会关注要不要给select指定nolock，这其中也包括自己没遇到过大数据吧，也包括自己的能力有限，只知道有锁这个玩意，细说的话就啥也不知道了，后来才知道携程有个规则，就是很多业务产线所写的select都必须指定nolock，懂一点的人可能会说nolock可以提升性能，如果你这样说，确实是这样，因为数据库的锁是有96字节开销的，没了锁，也就没有你在profile中看到accquired和released痉挛了，当你看完我的事件之后，你可能会意识到，性能提升不是最关心的，最关心就是不要出现死锁，锁等待。。。好了，言归正传，下面我们看看到底在数据库中可以指定多少个锁？？？

一：到底可以指定多少个锁

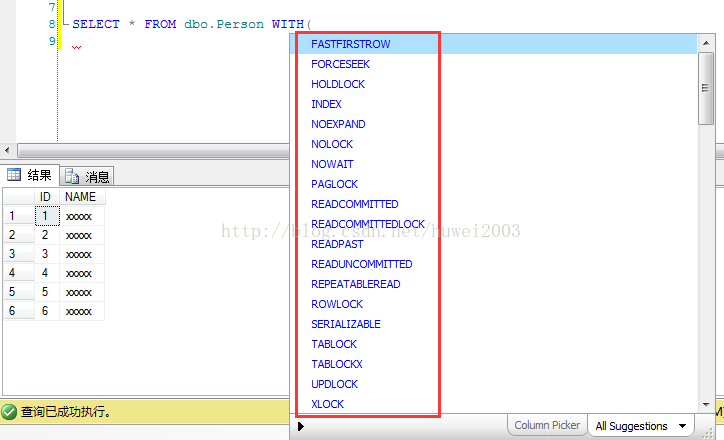
　　这个问题有意思，我们不需要记，只要你装一个SQL Prompt，有了这个神器，你就知道到底有多少个？如下图：

1 DROP TABLE dbo.Person

2 CREATE TABLE Person(ID INT IDENTITY,NAME CHAR(**4000**) DEFAULT 'xxxxx')

3 INSERT INTO dbo.Person DEFAULT VALUES

4 go **6**



一眼扫下去，还是蛮多的，不过你要注意了，那些所谓的XXXLock才是我们需要关注的，根据上面的图，我们大概把锁分个类。。。

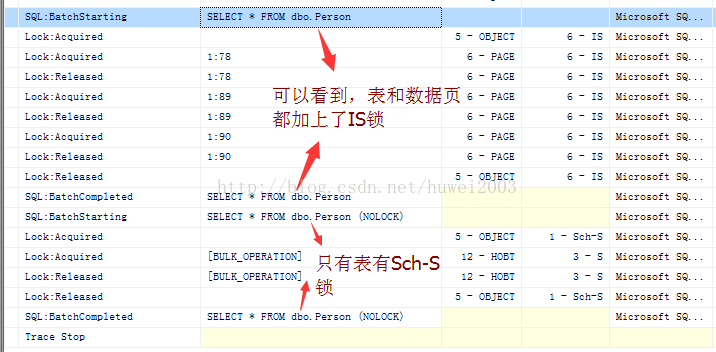
粒度锁：PAGLOCK, TABLOCK, TABLOCKX, ROWLOCK, NOLOCK

模式锁：HOLDLOCK, UPDLOCK, XLOCK

接下来我从粒度锁说起：

1. NOLOCK

　　都说nolock是无锁模式的，那到底是怎样的无锁呢？？？到这篇为止，你应该知道，如果不加nolock，我们的表，数据页是附加IS锁的，那接下来我用profile看下两者有什么区别。

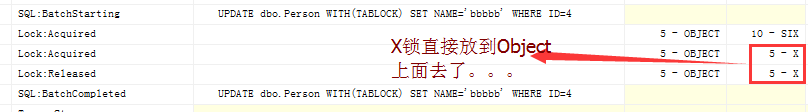


从上图中，你会看到加上nolock之后，object上面附加了Sch-S锁，这个锁叫做“架构稳定锁”，很简单就是sql编译时附加的一把锁，目的就是防止在编译时，有其他连接修改表结构，而这个锁只与Sch-M锁冲突，与其他锁都兼容，这说明什么？说明其他连接锁住了记录也没关系，我的nolock不跟他们打交道，这样的话，就可能会读到脏数据，不过没关系，携程的很多业务是容许脏数据的，毕竟比锁等待，死锁要强得多，再说nolock读到了其他连接未修改或者未提交的数据，这个概率也比较低，就算遇到了也没关系，一般不会招来客诉的，客人或许再刷下页面，数据或许就正确了，对不对。。。

2.TABLOCK

　 这个还是比较见名识义的，就是附加在table上的锁，也就是表锁了，很恐怖的。。。下面我举个Update的例子，看看前后对比。





在上面你有没有看到，X锁已经附加到OBJECT上面去了。。。这样的话，其他连接就动不了这个Object了，只能等待。。。

3.  PAGLOCK

　　看了名字你应该也知道，就是附加到页面这个级别的锁，我也举一个Update的例子。

1 BEGIN TRAN

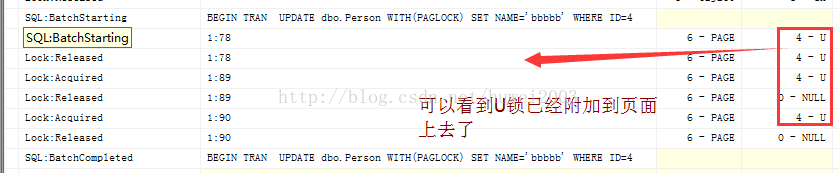
2 UPDATE dbo.Person SET NAME='aaaaa' WHERE ID=**6**

3

4 BEGIN TRAN

5 UPDATE dbo.Person WITH(PAGLOCK) SET NAME='bbbbb' WHERE ID=**4**





从上面两个图中，你应该可以看到，原来附加到RID上面的U锁，由于PagLock的提升，现在要附加到Page上面了，这个就是所谓的数据页锁。

4.TABLOCKX, ROWLOCK

   这两个我就不细说了，TABLOCKX就是直接附加在table上的X锁，你可以通过select看一下。



ROWLOCK的话，默认情况下就是ROWLOCK，比如默认的Update，你会发现RID上被附加的U锁，这个就是行锁。

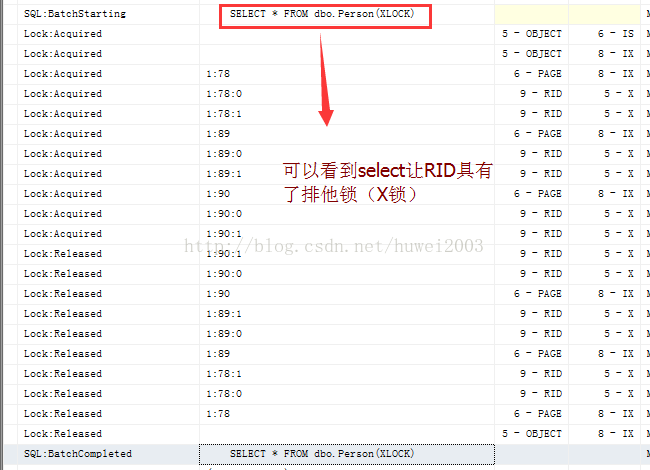
5.UPDLOCK

　这个锁还是蛮有意思的，它就是update锁，如果你select下，它会呈现update的锁痉挛效果。



6. XLOCK

　　知道了UPDLOCK锁，我想XLOCK你也应该明白了。。。它就是delete锁，即排他锁，我可以让select带上排他锁。

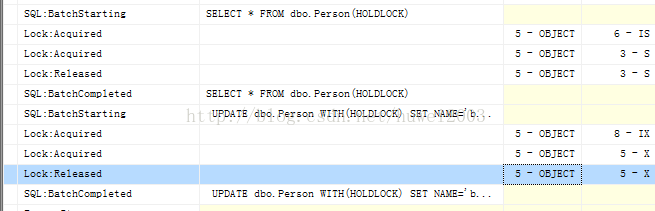


7.HOLDLOCK

　　最后一个我也没闹明白，据说是让语句在整个事务中持有锁，然后我就用select和update调试一下。

1 SELECT \* FROM dbo.Person(HOLDLOCK)

2 UPDATE dbo.Person WITH(HOLDLOCK) SET NAME='bbbbb' WHERE ID=**4**



## 三 SQL Server 锁机制 悲观锁 乐观锁 实测解析

在使用SQL时，大都会遇到这样的问题，你Update一条记录时，需要通过Select来检索出其值或条件，然后在通过这个值来执行修改操作。  
  
但当以上操作放到多线程中并发处理时会出现问题：某线程select了一条记录但还没来得及update时，另一个线程仍然可能会进来select到同一条记录。  
  
 一般解决办法就是使用锁和事物的联合机制：  
  
  
1. 把select放在事务中, 否则select完成, 锁就释放了  
2. 要阻止另一个select , 则要手工加锁, select 默认是共享锁, select之间的共享锁是不冲突的, 所以, 如果只是共享锁, 即使锁没有释放, 另一个select一样可以下共享锁, 从而select出数据

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. BEGIN TRAN
2. SELECT \* FROM Table WITH(UPDLOCK)
3. --或者 SELECT \* FROM Table WITH(TABLOCKX, READPAST) 具体情况而定。
4. UPDATE ....
5. COMMIT TRAN

所有Select加 With (NoLock)解决阻塞死锁,在查询语句中使用 NOLOCK 和 READPAST   
处理一个数据库死锁的异常时候，其中一个建议就是使用 NOLOCK 或者 READPAST 。有关 NOLOCK 和 READPAST的一些技术知识点：   
对于非银行等严格要求事务的行业，搜索记录中出现或者不出现某条记录，都是在可容忍范围内，所以碰到死锁，应该首先考虑，我们业务逻辑是否能容忍出现或者不出现某些记录，而不是寻求对双方都加锁条件下如何解锁的问题。   
NOLOCK 和 READPAST 都是处理查询、插入、删除等操作时候，如何应对锁住的数据记录。但是这时候一定要注意NOLOCK 和 READPAST的局限性，确认你的业务逻辑可以容忍这些记录的出现或者不出现：   
简单来说：  
  
1.NOLOCK 可能把没有提交事务的数据也显示出来  
2.READPAST 会把被锁住的行不显示出来  
  
不使用 NOLOCK 和 READPAST ，在 Select 操作时候则有可能报错误：事务(进程 ID \*\*)与另一个进程被死锁在 锁 资源上，并且已被选作死锁牺牲品。  
  
SELECT \* FROM Table WITH(NOLOCK)  
SELECT \* FROM Table WITH(READPAST)  
  
实际开始动手用代码说话吧！  
  
SQLServer2012在查询分析器里面开两个连接

#### 插入锁：

结论：“表锁”锁定对该表的Select、Update、Delete操作，但不影响对该表的Insert操作也不影响以主键Id为条件的Select，所以Select如果不想等待就要在Select后加With(Nolock)，但这样会产生脏数据就是其他事务已更新但并没有提交的数据，如果该事务进行了RollBack则取出的数据就是错误的，所以好自己权衡利弊，一般情况下90%以上的Select都允许脏读，只有账户金额相关的不允许。

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. ------------------A连接 Insert Lock-------------------
2. BEGIN TRAN
3. INSERT INTO dbo.UserInfo
4. ( Name, Age, Mobile, AddTime, Type )
5. VALUES  ( 'eee', -- Name - varchar(50)
6. 2, -- Age - int
7. '555', -- Mobile - char(11)
8. GETDATE(), -- AddTime - datetime
9. 0  -- Type - int
10. )
12. SELECT resource\_type, request\_mode,COUNT(\*)  FROM sys.dm\_tran\_locks
13. WHERE request\_session\_id=@@SPID
14. GROUP BY resource\_type,request\_mode
15. --ROLLBACK TRAN
17. ------------------------B连接 Insert Lock------------------------
18. INSERT INTO dbo.UserInfo
19. ( Name, Age, Mobile, AddTime, Type )
20. VALUES  ( 'fff', -- Name - varchar(50)
21. 2, -- Age - int
22. '123', -- Mobile - char(11)
23. GETDATE(), -- AddTime - datetime
24. 1  -- Type - int
25. ) --可以执行插入
27. SELECT \* FROM dbo.UserInfo --需要等待解锁
28. SELECT \* FROM dbo.UserInfo WHERE Age=1 --需要等待解锁
29. SELECT \* FROM dbo.UserInfo WHERE Id=3 --可以执行查询（根据主键可以）
30. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) --可以执行查询(在一个事务中，有更新字段但还没有提交，此时就会查处脏数据)
31. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) WHERE Age=1 --可以执行查询
32. UPDATE dbo.UserInfo SET Type=5 WHERE Name='fff'  --需要等待解锁
33. DELETE FROM dbo.UserInfo WHERE Name='fff' --需要等待解锁

#### 更新锁：

结论：“表锁”锁定对该表的Select、Update、Delete操作，但不影响对该表的Insert操作也不影响以主键Id为条件的Select

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. -----------------------A连接 Update Lock-----------------------
2. BEGIN TRAN
3. UPDATE dbo.UserInfo SET Name = 'eee' WHERE Age = 2
5. SELECT resource\_type, request\_mode,COUNT(\*)  FROM sys.dm\_tran\_locks
6. WHERE request\_session\_id=@@SPID
7. GROUP BY resource\_type,request\_mode
9. --ROLLBACK TRAN
11. ------------------------B连接 Update Lock------------------------
12. INSERT INTO dbo.UserInfo
13. ( Name, Age, Mobile, AddTime, Type )
14. VALUES  ( 'ppp', -- Name - varchar(50)
15. 15, -- Age - int
16. '666', -- Mobile - char(11)
17. GETDATE(), -- AddTime - datetime
18. 9  -- Type - int
19. ) --可以执行插入
20. SELECT \* FROM dbo.UserInfo --需要等待解锁
21. SELECT \* FROM dbo.UserInfo WHERE Name='ppp' --需要等待解锁
22. SELECT \* FROM dbo.UserInfo WHERE Id=3 --可以执行查询（根据主键可以）
23. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) --可以执行查询(在一个事务中，有更新字段但还没有提交，此时就会查处脏数据)
24. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) WHERE Name = 'ppp' --可以执行查询
25. UPDATE dbo.UserInfo SET Age=8 WHERE Name='ccc' --需要等待解锁
26. DELETE dbo.UserInfo WHERE Age = 5 --需要等待解锁

#### 主键锁：

结论：“行锁+表锁” 锁定对该表的Select、Update、Delete操作，但不影响对该表的Insert操作也不影响以主键Id为条件的Select、Update、Delete

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. ------------------------A连接 Key Lock--------------------
2. BEGIN TRAN
3. UPDATE dbo.UserInfo SET Name='hhh' WHERE Id=3 --以主键为条件
5. SELECT resource\_type, request\_mode,COUNT(\*)  FROM sys.dm\_tran\_locks
6. WHERE request\_session\_id=@@SPID
7. GROUP BY resource\_type,request\_mode
9. --ROLLBACK TRAN
11. ------------------------B连接 Key Lock----------------------
12. INSERT INTO dbo.UserInfo
13. ( Name, Age, Mobile, AddTime, Type )
14. VALUES  ( 'kkk', -- Name - varchar(50)
15. 18, -- Age - int
16. '234', -- Mobile - char(11)
17. GETDATE(), -- AddTime - datetime
18. 7  -- Type - int
19. ) --可以执行插入
20. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) --可以执行查询(在一个事务中，有更新字段但还没有提交，此时就会查处脏数据)
21. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) WHERE Name = 'kkk' --可以执行查询
23. -----//全表查询及操作正在处理的行
24. SELECT \* FROM dbo.UserInfo --需要等待解锁
25. SELECT \* FROM dbo.UserInfo WHERE Id=3 --需要等待解锁（根据主键,但与A连接操作相同行不可）
26. UPDATE dbo.UserInfo SET Name='mmm' WHERE Id=3 --需要等待解锁（根据主键,但与A连接操作相同行不可）
27. DELETE dbo.UserInfo WHERE Id=3 --需要等待解锁（根据主键,但与A连接操作相同行不可）
28. -----//使用非主键为条件的操作
29. SELECT \* FROM dbo.UserInfo WHERE Name='aaa' --需要等待解锁（非主键不可）
30. UPDATE dbo.UserInfo SET Name='ooo' WHERE Name='aaa' --需要等待解锁（非主键不可）
31. DELETE dbo.UserInfo WHERE Name='aaa' --需要等待解锁（非主键不可）
32. -----//使用主键为条件的操作
33. SELECT \* FROM dbo.UserInfo WHERE id=1 --可以执行查询（根据主键可以）
34. UPDATE dbo.UserInfo SET Name='yyy' WHERE Id=1 --可以执行更新（根据主键可以）
35. DELETE dbo.UserInfo WHERE Id=1 --可以执行删除（根据主键可以）

#### 索引锁：

结论：“行锁+表锁” 锁定对该表的Select、Update、Delete操作，但不影响对该表的Insert操作也不影响以主键Id为条件的Select、Update、Delete，也不影响以索引列Name为条件的Update、Delete但不可以Select

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. ------------------------A连接 Index Lock--------------------
2. DROP INDEX dbo.UserInfo.Index\_UserInfo\_Name
3. CREATE INDEX Index\_UserInfo\_Name ON dbo.UserInfo(Name)
5. BEGIN TRAN
6. UPDATE dbo.UserInfo SET age=66 WHERE Name='ddd' --使用name索引列为条件
8. SELECT resource\_type, request\_mode,COUNT(\*)  FROM sys.dm\_tran\_locks
9. WHERE request\_session\_id=@@SPID
10. GROUP BY resource\_type,request\_mode
12. --ROLLBACK TRAN
14. ----------------------B连接 Index Lock-------------------
15. INSERT INTO dbo.UserInfo
16. ( Name, Age, Mobile, AddTime, Type )
17. VALUES  ( 'iii', -- Name - varchar(50)
18. 20, -- Age - int
19. '235235235', -- Mobile - char(11)
20. GETDATE(), -- AddTime - datetime
21. 12  -- Type - int
22. ) --可以执行插入
23. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) --可以执行查询(在一个事物中，有更新字段但还没有提交，此时就会查处脏数据)
24. SELECT \* FROM dbo.UserInfo WITH(NOLOCK) WHERE Name = 'kkk' --可以执行查询
26. -----//全表查询及操作正在处理的行
27. SELECT \* FROM dbo.UserInfo --需要等待解锁
28. SELECT \* FROM dbo.UserInfo WHERE Id=4 --需要等待解锁（根据主键,但与A连接操作相同行不可）
29. UPDATE dbo.UserInfo SET Name='mmm' WHERE Id=4 --需要等待解锁（根据主键,但与A连接操作相同行不可）
30. DELETE dbo.UserInfo WHERE Id=4 --需要等待解锁（根据主键,但与A连接操作相同行不可）
31. -----//使用非主键非索引为条件的操作
32. SELECT \* FROM dbo.UserInfo WHERE Age=5 --需要等待解锁（非主键不可）
33. UPDATE dbo.UserInfo SET Name='ooo' WHERE Age=5 --需要等待解锁（非主键不可）
34. DELETE dbo.UserInfo WHERE Age=5 --需要等待解锁（非主键不可）
35. -----//使用主键为条件的操作
36. SELECT \* FROM dbo.UserInfo WHERE Id=1 --可以执行更新（根据主键可以）
37. UPDATE dbo.UserInfo SET Name='yyy' WHERE Id=1 --可以执行更新（根据主键可以）
38. DELETE dbo.UserInfo WHERE Id=1 --可以执行删除（根据主键可以）
39. -----//使用索引为条件的操作
40. SELECT \* FROM dbo.UserInfo WHERE Name='aaa' --需要等待解锁（非主键不可）
41. UPDATE dbo.UserInfo SET Name='ooo' WHERE Name='aaa' --可以执行更新（根据索引可以）
42. DELETE dbo.UserInfo WHERE Name='aaa' --可以执行删除（根据索引可以）

#### 悲观锁（更新锁-人工手动设置上锁）：

结论：可以理解为在使用版本控制软件的时候A迁出了一个文件，并且8i将这个87文件锁定，B就无法再迁出该文件了，直到A迁入解锁后才能被其他人迁出。

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. ------------------------A连接 Update Lock(悲观锁)---------------------
2. BEGIN TRAN
3. SELECT \* FROM dbo.UserInfo WITH(UPDLOCK) WHERE Id=2
5. SELECT resource\_type, request\_mode,COUNT(\*)  FROM sys.dm\_tran\_locks
6. WHERE request\_session\_id=@@SPID
7. GROUP BY resource\_type,request\_mode
9. --COMMIT TRAN
10. --ROLLBACK TRAN
12. ---------------------------B连接 Update Lock(悲观锁)-------------------------
13. SELECT \* FROM dbo.UserInfo --可以执行查询
14. SELECT \* FROM dbo.UserInfo WHERE id=2 --可以执行查询
15. SELECT \* FROM dbo.UserInfo WHERE Name='ooo' --可以执行查询
17. UPDATE dbo.UserInfo SET Age=3 WHERE id=1 --可以执行更新（根据主键可以）
18. UPDATE dbo.UserInfo SET Age=3 WHERE Name='ccc' --需要等待解锁（非主键不可）
20. DELETE dbo.UserInfo WHERE id=1 --可以执行更新（根据主键可以）
21. DELETE dbo.UserInfo WHERE name='ccc' --需要等待解锁（非主键不可）

#### 乐观锁（人工通过逻辑在数据库中模拟锁）

结论：可以理解为同样在使用版本控制软件的时候A迁出了一个文件，B也可以迁出该文件，两个人都可以对此文件进行修改，其中一个人先进行提交的时候，版本并没有变化所以可以正常提交，另一个后提交的时候，发现版本增加不对称了，就提示冲突由用户来选择如何进行合并再重新进行提交。

**[sql]** [view plain](http://blog.csdn.net/huwei2003/article/details/4047191) [copy](http://blog.csdn.net/huwei2003/article/details/4047191)

1. --------------------------A客户端连接 Lock(乐观锁)------------------------
2. --DROP TABLE Coupon
3. -----------------创建优惠券表-----------------
4. CREATE TABLE Coupon
5. (
6. Id INT PRIMARY KEY IDENTITY(1,1),
7. Number VARCHAR(50) NOT NULL,
8. [User] VARCHAR(50),
9. UseTime DATETIME,
10. IsFlag BIT DEFAULT(0) NOT NULL,
11. CreateTime DATETIME DEFAULT(GETDATE()) NOT NULL
12. )
13. INSERT INTO dbo.Coupon(Number) VALUES ( '10000001')
14. INSERT INTO dbo.Coupon(Number) VALUES ( '10000002')
15. INSERT INTO dbo.Coupon(Number) VALUES ( '10000003')
16. INSERT INTO dbo.Coupon(Number) VALUES ( '10000004')
17. INSERT INTO dbo.Coupon(Number) VALUES ( '10000005')
18. INSERT INTO dbo.Coupon(Number) VALUES ( '10000006')
20. --SELECT \* FROM dbo.Coupon WITH(NOLOCK) --查询数据
21. --UPDATE Coupon SET [User]=NULL, UseTime=NULL, IsFlag=0 --还原数据
23. -----------------1、模拟高并发普通更新-----------------
24. DECLARE @User VARCHAR(50)    --模拟要使用优惠券的用户
25. DECLARE @TempId INT            --模拟抽选出来的要使用的优惠券
26. SET @User='a'
27. BEGIN TRAN
28. SELECT @TempId=Id FROM dbo.Coupon WHERE IsFlag=0    --高并发时此语句有可能另外一个该事务已取出的Id
29. --WAITFOR DELAY '00:00:05'    --改用此方式要开两个SQL Management客户端
30. UPDATE dbo.Coupon SET IsFlag=1, [User]=@User, UseTime=GETDATE() WHERE Id=@TempId
31. COMMIT TRAN
32. --ROLLBACK TRAN
34. -----------------2、悲观锁解决方案-----------------
35. DECLARE @User VARCHAR(50)    --模拟要使用优惠券的用户
36. DECLARE @TempId INT            --模拟抽选出来的要使用的优惠券
37. SET @User='a'
38. BEGIN TRAN
39. SELECT @TempId=Id FROM dbo.Coupon WITH(UPDLOCK) WHERE IsFlag=0    --高并发时此语句会锁定取出的Id数据行
40. --WAITFOR DELAY '00:00:05'    --改用此方式要开两个SQL Management客户端
41. UPDATE dbo.Coupon SET IsFlag=1, [User]=@User, UseTime=GETDATE() WHERE Id=@TempId
42. COMMIT TRAN
43. --ROLLBACK TRAN
45. -----------------3、乐观锁解决方案-----------------
46. ALTER TABLE dbo.Coupon ADD RowVer ROWVERSION NOT NULL --增加数据行版本戳类型字段（微软新推荐数据字段，该字段每张表只能有一个，会在创建行或更新行时自动进行修改无需人为干涉，该字段不能建立索引及主键因为会频繁修改）

49. DECLARE @User VARCHAR(50)    --模拟要使用优惠券的用户
50. DECLARE @TempId INT            --模拟抽选出来的要使用的优惠券
51. DECLARE @RowVer BINARY(8)    --抽选出来的优惠券的版本（ROWVERSION数据类型存储大小为8字节）
52. SET @User='a'
54. BEGIN TRY
55. BEGIN TRAN
56. SELECT @TempId=Id, @RowVer=RowVer FROM dbo.Coupon WHERE IsFlag=0    --取出可用的Id及对应的版本戳
57. --WAITFOR DELAY '00:00:05'    --改用此方式要开两个SQL Management客户端
58. UPDATE dbo.Coupon SET IsFlag=1, [User]=@User, UseTime=GETDATE() WHERE Id=@TempId AND RowVer=@RowVer
59. IF(@@ROWCOUNT > 0)
60. BEGIN
61. PRINT('修改成功')
62. COMMIT TRAN
63. END
64. ELSE
65. BEGIN
66. PRINT('该数据已被其他用户修改')
67. ROLLBACK TRAN
68. END
69. END TRY
70. BEGIN CATCH
71. ROLLBACK TRAN
72. END CATCH
74. --------------------------B客户端连接 Lock(乐观锁)------------------------
75. --此测试需要开两个SQL Management Studio客户端，在A客户端使用WAITFOR DELAY来模拟并发占用，在B客户端执行与A客户端相同的SQL脚本即可（注释掉WAITFOR），所以在此不放相同代码了。

在乐观锁和悲观锁之间进行选择的标准是：冲突的频率与严重性。如果冲突很少，或者冲突的后果不会很严重，那么通常情况下应该选择乐观锁，因为它能得到更好的并发性，而且更容易实现。但是，如果冲突的结果对于用户来说痛苦的，那么就需要使用悲观策略。  
  
我认为如果同一张表的并发很高，但并发处理同一条数据的冲突几率很低，那就应该使用乐观锁，反之，如果同一张表的并发不高，但同时处理同一条数据的几率很高，就应该使用悲观锁。

## 四 SQL Server 中WITH (NOLOCK)浅析

### 概念介绍

开发人员喜欢在SQL脚本中使用WITH(NOLOCK), WITH(NOLOCK)其实是表提示（table\_hint）中的一种。它等同于 READUNCOMMITTED 。 具体的功能作用如下所示（摘自MSDN）：

   1： 指定允许脏读。不发布共享锁来阻止其他事务修改当前事务读取的数据，其他事务设置的排他锁不会阻碍当前事务读取锁定数据。允许脏读可能产生较多的并发操作，但其代价是读取以后会被其他事务回滚的数据修改。这可能会使您的事务出错，向用户显示从未提交过的数据，或者导致用户两次看到记录（或根本看不到记录）。有关脏读、不可重复读和幻读的详细信息，请参阅[并发影响](http://msdn.microsoft.com/zh-cn/library/ms190805(v=sql.105).aspx)。

   2： READUNCOMMITTED 和 NOLOCK 提示仅适用于数据锁。所有查询（包括那些带有 READUNCOMMITTED 和 NOLOCK 提示的查询）都会在编译和执行过程中获取 Sch-S（架构稳定性）锁。因此，当并发事务持有表的 Sch-M（架构修改）锁时，将阻塞查询。例如，数据定义语言 (DDL) 操作在修改表的架构信息之前获取 Sch-M 锁。所有并发查询（包括那些使用 READUNCOMMITTED 或 NOLOCK 提示运行的查询）都会在尝试获取 Sch-S 锁时被阻塞。相反，持有 Sch-S 锁的查询将阻塞尝试获取 Sch-M 锁的并发事务。有关锁行为的详细信息，请参阅[锁兼容性（数据库引擎）](http://msdn.microsoft.com/zh-cn/library/ms186396(v=sql.105).aspx)。

   3:  不能为通过插入、更新或删除操作修改过的表指定 READUNCOMMITTED 和 NOLOCK。SQL Server 查询优化器忽略 FROM 子句中应用于 UPDATE 或 DELETE 语句的目标表的 READUNCOMMITTED 和 NOLOCK 提示。

功能与缺陷

    使用WIHT(NOLOCK)有利也有弊，所以在决定使用之前，你一定需要了解清楚WITH(NOLOCK)的功能和缺陷，看其是否适合你的业务需求,不要觉得它能提升性能，稀里糊涂的就使用它。

    1：使用WITH(NOLOCK)时查询不受其它排他锁阻塞

    打开会话窗口1，执行下面脚本，不提交也不回滚事务，模拟事务真在执行过程当中

BEGIN TRAN

UPDATE TEST SET NAME='Timmy' WHERE OBJECT\_ID =1;

--ROLLBACK

   打开会话窗口2，执行下面脚本，你会发现执行结果一直查询不出来（其实才两条记录）。当前会话被阻塞了

SELECT \* FROM TEST;

    打开会话窗口3，执行下面脚本，查看阻塞情况，你会发现在会话2被会话1给阻塞了，会话2的等待类型为LCK\_M\_S：“当某任务正在等待获取共享锁时出现”

SELECT wt.blocking\_session\_id AS BlockingSessesionId

,sp.program\_name AS ProgramName

,COALESCE(sp.LOGINAME, sp.nt\_username) AS HostName

,ec1.client\_net\_address AS ClientIpAddress

,db.name AS DatabaseName

,wt.wait\_type AS WaitType

,ec1.connect\_time AS BlockingStartTime

,wt.WAIT\_DURATION\_MS/1000 AS WaitDuration

,ec1.session\_id AS BlockedSessionId

,h1.TEXT AS BlockedSQLText

,h2.TEXT AS BlockingSQLText

FROM sys.dm\_tran\_locks AS tl

INNER JOIN sys.databases db

ON db.database\_id = tl.resource\_database\_id

INNER JOIN sys.dm\_os\_waiting\_tasks AS wt

ON tl.lock\_owner\_address = wt.resource\_address

INNER JOIN sys.dm\_exec\_connections ec1

ON ec1.session\_id = tl.request\_session\_id

INNER JOIN sys.dm\_exec\_connections ec2

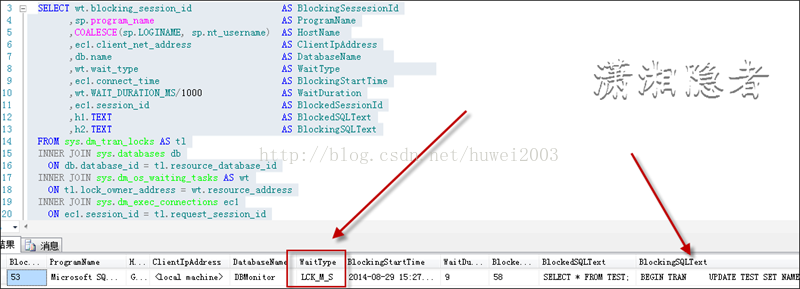
ON ec2.session\_id = wt.blocking\_session\_id

LEFT OUTER JOIN master.dbo.sysprocesses sp

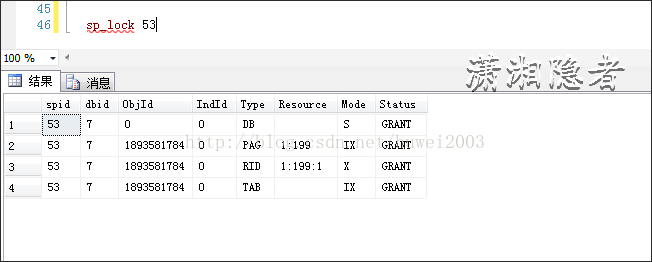
ON SP.spid = wt.blocking\_session\_id

CROSS APPLY sys.dm\_exec\_sql\_text(ec1.most\_recent\_sql\_handle) AS h1

CROSS APPLY sys.dm\_exec\_sql\_text(ec2.most\_recent\_sql\_handle) AS h2



此时查看会话1（会话1的会话ID为53，执行脚本1前，可以用SELECT  @@spid查看会话ID）的锁信息情况，你会发现表TEST(ObjId=1893581784)持有的锁信息如下所示



打开会话窗口4，执行下面脚本.你会发现查询结果很快就出来，会话4并不会被会话1阻塞。

    SELECT \* FROM TEST WITH(NOLOCK)

从上面模拟的这个小例子可以看出，正是由于加上WITH(NOLOCK)提示后，会话1中事务设置的排他锁不会阻碍当前事务读取锁定数据，所以会话4不会被阻塞，从而提升并发时查询性能。

2：WITH(NOLOCK) 不发布共享锁来阻止其他事务修改当前事务读取的数据，这个就不举例子了。

本质上WITH(NOLOCK)是通过减少锁和不受排它锁影响来减少阻塞，从而提高并发时的性能。所谓凡事有利也有弊，WITH(NOLOCK)在提升性能的同时，也会产生脏读现象。

如下所示，表TEST有两条记录，我准备更新OBJECT\_ID=1的记录，此时事务既没有提交也没有回滚



BEGIN TRAN

UPDATE TEST SET NAME='Timmy' WHERE OBJECT\_ID =1;

--ROLLBACK

此时另外一个会话使用WITH(NOLOCK)查到的记录为未提交的记录值



假如由于某种原因，该事务回滚了，那么我们读取到的OBJECT\_ID=1的记录就是一条脏数据。

脏读又称无效数据的读出，是指在数据库访问中，[事务](http://baike.baidu.com/view/121511.htm)T1将某一值修改，然后事务T2读取该值，此后T1因为某种原因撤销对该值的修改，这就导致了T2所读取到的数据是无效的。

### WITH(NOLOCK)使用场景

什么时候可以使用WITH(NOLOCK)? 什么时候不能使用WITH(NOLOCK),这个要视你系统业务情况，综合考虑性能情况与业务要求来决定是否使用WITH(NOLOCK), 例如涉及到金融或会计成本之类的系统，出现脏读那是要产生严重问题的。关键业务系统也要慎重考虑。大体来说一般有下面一些场景可以使用WITH(NOLOCK)

   1: 基础数据表，这些表的数据很少变更。

   2：历史数据表，这些表的数据很少变更。

   3：业务允许脏读情况出现涉及的表。

   4：数据量超大的表，出于性能考虑，而允许脏读。

另外一点就是不要滥用WITH(NOLOCK),我发现有个奇怪现象，很多开发知道WITH(NOLOCK),但是有不了解脏读，习惯性的使用WITH(NOLOCK)。

### WITH(NOLOCK)与 NOLOCK区别

为了搞清楚WITH(NOLOCK)与NOLOCK的区别，我查了大量的资料，我们先看看下面三个SQL语句有啥区别

    SELECT \* FROM TEST NOLOCK

    SELECT \* FROM TEST (NOLOCK);

    SELECT \* FROM TEST WITH(NOLOCK);

上面的问题概括起来也就是说NOLOCK、(NOLOCK)、 WITH(NOLOCK)的区别：

1： NOLOCK这样的写法，其实NOLOCK其实只是别名的作用，而没有任何实质作用。所以不要粗心将(NOLOCK)写成NOLOCK

2：（NOLOCK）与WITH(NOLOCK)其实功能上是一样的。(NOLOCK)只是WITH(NOLOCK)的别名,但是在SQL Server 2008及以后版本中，(NOLOCK)不推荐使用了，"不借助 WITH 关键字指定表提示”的写法已经过时了。 具体参见MSDN <http://msdn.microsoft.com/zh-cn/library/ms143729>%28SQL.100%29.aspx

    2.1  至于网上说WITH(NOLOCK)在SQL SERVER 2000不生效，我验证后发现完全是个谬论。

    2.2  在使用链接服务器的SQL当中，(NOLOCK)不会生效，WITH(NOLOCK)才会生效。如下所示



    消息 4122，级别 16，状态 1，第 1 行

    Remote table-valued function calls are not allowed.

3.语法上有些许出入，如下所示

这种语法会报错

SELECT  \* FROM   sys.indexes  WITH(NOLOCK) AS i

-Msg 156, Level 15, State 1, Line 1

-Incorrect syntax near the keyword 'AS'.

这种语法正常

SELECT  \* FROM   sys.indexes  (NOLOCK) AS i

可以全部改写为下面语法

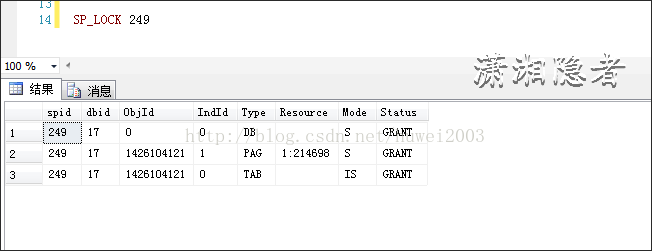
SELECT  \* FROM   sys.indexes   i WITH(NOLOCK)

SELECT  \* FROM   sys.indexes   i (NOLOCK)

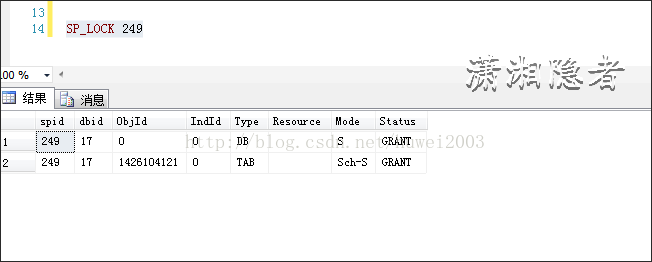
### WITH(NOLOCK)会不会产生锁

    很多人误以为使用了WITH(NOLOCK)后，数据库库不会产生任何锁。实质上，使用了WITH(NOLOCK)后，数据库依然对该表对象生成Sch-S（架构稳定性）锁以及DB类型的共享锁， 如下所示，可以在一个会话中查询一个大表，然后在另外一个会话中查看锁信息(也可以使用SQL Profile查看会话锁信息)

    不使用WTIH(NOLOCK)



  使用WITH(NOLOCK)



  从上可以看出使用WITH(NOLOCK)后，数据库并不是不生成相关锁。  对比可以发现使用WITH(NOLOCK)后，数据库只会生成DB类型的共享锁、以及TAB类型的架构稳定性锁.

另外，使用WITH(NOLOCK)并不是说就不会被其它会话阻塞，依然可能会产生Schema Change Blocking

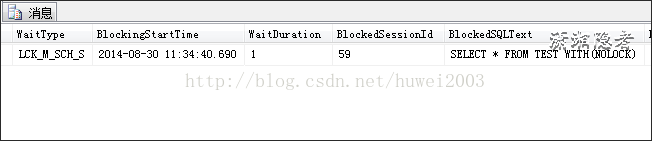
会话1：执行下面SQL语句，暂时不提交，模拟事务正在执行

BEGIN TRAN

ALTER TABLE TEST ADD Grade VARCHAR(10) ;

会话2：执行下面语句，你会发现会话被阻塞，截图如下所示。

SELECT \* FROM TEST WITH(NOLOCK)



-----

本文引用了以下网址内容：

http://www.cnblogs.com/huangxincheng/p/4292320.html

http://blog.itpub.net/13651903/viewspace-1091664/

http://www.cnblogs.com/kerrycode/p/3946268.html

http://www.cnblogs.com/taiyonghai/p/5674462.html

--- end ---

<https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx>

# SQL Server Transaction Locking and Row Versioning Guide

In any database, mismanagement of transactions often leads to contention and performance problems in systems that have many users. As the number of users that access the data increases, it becomes important to have applications that use transactions efficiently. This guide describes the locking and row versioning mechanisms the SQL Server Database Engine uses to ensure the physical integrity of each transaction and provides information on how applications can control transactions efficiently.

|  |
| --- |
| Applies to: SQL Server 2005 through SQL Server 2012 unless noted otherwise. |

## In This Guide

[Transaction Basics](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Basics)

[Locking and Row Versioning Basics](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Lock_Basics)

[Locking in the Database Engine](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Lock_Engine)

[Row Versioning-based Isolation Levels in the Database Engine](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Row_versioning)

[Customizing Locking and Row Versioning](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Customize)

[Advanced Transaction Information](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Advanced)

## Transaction Basics

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

Atomicity

A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them are performed.

Consistency

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

Isolation

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

Durability

After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

SQL programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data. The programmer must define the sequence of data modifications that leave the data in a consistent state relative to the organization's business rules. The programmer includes these modification statements in a single transaction so that the SQL Server Database Engine can enforce the physical integrity of the transaction.

It is the responsibility of an enterprise database system, such as an instance of the Database Engine, to provide mechanisms ensuring the physical integrity of each transaction. The Database Engine provides:

* Locking facilities that preserve transaction isolation.
* Logging facilities that ensure transaction durability. Even if the server hardware, operating system, or the instance of the Database Engine itself fails, the instance uses the transaction logs upon restart to automatically roll back any uncompleted transactions to the point of the system failure.
* Transaction management features that enforce transaction atomicity and consistency. After a transaction has started, it must be successfully completed (committed), or the Database Engine undoes all of the data modifications made since the transaction started. This operation is referred to as rolling back a transaction because it returns the data to the state it was prior to those changes.

### Controlling Transactions

Applications control transactions mainly by specifying when a transaction starts and ends. This can be specified by using either Transact-SQL statements or database application programming interface (API) functions. The system must also be able to correctly handle errors that terminate a transaction before it completes. For more information, see [Transaction Statements (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms174377(v=sql.110).aspx), [Transactions in ODBC](https://technet.microsoft.com/library/ms131281.aspx) and [Transactions in SQL Server Native Client (OLEDB)](https://msdn.microsoft.com/en-us/library/ms130918.aspx).

By default, transactions are managed at the connection level. When a transaction is started on a connection, all Transact-SQL statements executed on that connection are part of the transaction until the transaction ends. However, under a multiple active result set (MARS) session, a Transact-SQL explicit or implicit transaction becomes a batch-scoped transaction that is managed at the batch level. When the batch completes, if the batch-scoped transaction is not committed or rolled back, it is automatically rolled back by SQL Server. For more information, see [Multiple Active Result Sets (MARS) in SQL Server](https://msdn.microsoft.com/library/ms345109(v=sql.90).aspx).

#### Starting Transactions

Using API functions and Transact-SQL statements, you can start transactions in an instance of the SQL Server Database Engine as explicit, autocommit, or implicit transactions.

Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and end of the transaction through an API function or by issuing the Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements. When the transaction ends, the connection returns to the transaction mode it was in before the explicit transaction was started, either implicit or autocommit mode.

You can use all Transact-SQL statements in an explicit transaction, except for the following statements:

|  |  |  |
| --- | --- | --- |
| ALTER DATABASE | CREATE DATABASE | DROP FULLTEXT INDEX |
| ALTER FULLTEXT CATALOG | CREATE FULLTEXT CATALOG | RECONFIGURE |
| ALTER FULLTEXT INDEX | CREATE FULLTEXT INDEX | RESTORE |
| BACKUP | DROP DATABASE | Full-text system stored procedures |
| CREATE DATABASE | DROP FULLTEXT CATALOG | sp\_dboption to set database options or any system procedure that modifies the master database inside explicit or implicit transactions. |
| **NoteNote** | | |
| UPDATE STATISTICS can be used inside an explicit transaction. However, UPDATE STATISTICS commits independently of the enclosing transaction and cannot be rolled back. | | |

Autocommit Transactions

Autocommit mode is the default transaction management mode of the SQL Server Database Engine. Every Transact-SQL statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A connection to an instance of the Database Engine operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Autocommit mode is also the default mode for ADO, OLE DB, ODBC, and DB-Library.

Implicit Transactions

When a connection is operating in implicit transaction mode, the instance of the Database Engine automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions. Set implicit transaction mode on through either an API function or the Transact-SQL SET IMPLICIT\_TRANSACTIONS ON statement.

After implicit transaction mode has been set on for a connection, the instance of the Database Engine automatically starts a transaction when it first executes any of these statements:

|  |  |  |
| --- | --- | --- |
| ALTER TABLE | FETCH | REVOKE |
| CREATE | GRANT | SELECT |
| DELETE | INSERT | TRUNCATE TABLE |
| DROP | OPEN | UPDATE |

Batch-scoped Transactions

Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

Distributed Transactions

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Each instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the Open Group XA specification for distributed transaction processing. For more information, see the MS DTC documentation.

A transaction within a single instance of the Database Engine that spans two or more databases is actually a distributed transaction. The instance manages the distributed transaction internally; to the user, it operates as a local transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

Prepare phase

When the transaction manager receives a commit request, it sends a prepare command to all of the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable, and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.

Commit phase

If the transaction manager receives successful prepares from all of the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all of the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

Database Engine applications can manage distributed transactions either through Transact-SQL or the database API. For more information, see [BEGIN DISTRIBUTED TRANSACTION (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms188386(v=sql.110).aspx).

#### Ending Transactions

You can end transactions with either a COMMIT or ROLLBACK statement, or through a corresponding API function.

COMMIT

If a transaction is successful, commit it. A COMMIT statement guarantees all of the transaction's modifications are made a permanent part of the database. A COMMIT also frees resources, such as locks, used by the transaction.

ROLLBACK

If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A ROLLBACK statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A ROLLBACK also frees resources held by the transaction.

|  |
| --- |
| **NoteNote** |
| Under connections enabled to support multiple active result sets (MARS), an explicit transaction started through an API function cannot be committed while there are pending requests for execution. Any attempt to commit this type of transaction while there are outstanding operations running will result in an error. |

#### Errors During Transaction Processing

If an error prevents the successful completion of a transaction, SQL Server automatically rolls back the transaction and frees all resources held by the transaction. If the client's network connection to an instance of the Database Engine is broken, any outstanding transactions for the connection are rolled back when the network notifies the instance of the break. If the client application fails or if the client computer goes down or is restarted, this also breaks the connection, and the instance of the Database Engine rolls back any outstanding connections when the network notifies it of the break. If the client logs off the application, any outstanding transactions are rolled back.

If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in the Database Engine is to roll back only the statement that generated the error. You can change this behavior using the SET XACT\_ABORT statement. After SET XACT\_ABORT ON is executed, any run-time statement error causes an automatic rollback of the current transaction. Compile errors, such as syntax errors, are not affected by SET XACT\_ABORT. For more information, see [SET XACT\_ABORT (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms188792(v=sql.110).aspx).

When errors occur, corrective action (COMMIT or ROLLBACK) should be included in application code. One effective tool for handling errors, including those in transactions, is the Transact-SQL TRY…CATCH construct. For more information with examples that include transactions, see [TRY...CATCH (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms175976(v=sql.110).aspx). Beginning with SQL Server 2012, you can use the THROW statement to raise an exception and transfers execution to a CATCH block of a TRY…CATCH construct. For more information, see [THROW (Transact-SQL)](https://technet.microsoft.com/en-us/library/ee677615(v=sql.110).aspx).

#### Compile and Run-time Errors in Autocommit mode

In autocommit mode, it sometimes appears as if an instance of the Database Engine has rolled back an entire batch instead of just one SQL statement. This happens if the error encountered is a compile error, not a run-time error. A compile error prevents the Database Engine from building an execution plan, so nothing in the batch is executed. Although it appears that all of the statements before the one generating the error were rolled back, the error prevented anything in the batch from being executed. In the following example, none of the INSERT statements in the third batch are executed because of a compile error. It appears that the first two INSERT statements are rolled back when they are never executed.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBatch VALUSE (3, 'ccc'); -- Syntax error.

GO

SELECT \* FROM TestBatch; -- Returns no rows.

GO

In the following example, the third INSERT statement generates a run-time duplicate primary key error. The first two INSERT statements are successful and committed, so they remain after the run-time error.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBatch VALUES (1, 'ccc'); -- Duplicate key error.

GO

SELECT \* FROM TestBatch; -- Returns rows 1 and 2.

GO

The Database Engine uses deferred name resolution, in which object names are not resolved until execution time. In the following example, the first two INSERT statements are executed and committed, and those two rows remain in the TestBatch table after the third INSERT statement generates a run-time error by referring to a table that does not exist.

CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3));

GO

INSERT INTO TestBatch VALUES (1, 'aaa');

INSERT INTO TestBatch VALUES (2, 'bbb');

INSERT INTO TestBch VALUES (3, 'ccc'); -- Table name error.

GO

SELECT \* FROM TestBatch; -- Returns rows 1 and 2.

GO

Arrow icon used with Back to Top link[[Top]](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Top)

## Locking and Row Versioning Basics

The SQL Server Database Engine uses the following mechanisms to ensure the integrity of transactions and maintain the consistency of databases when multiple users are accessing data at the same time:

* Locking

Each transaction requests locks of different types on the resources, such as rows, pages, or tables, on which the transaction is dependent. The locks block other transactions from modifying the resources in a way that would cause problems for the transaction requesting the lock. Each transaction frees its locks when it no longer has a dependency on the locked resources.

* Row versioning

When a row versioning-based isolation level is enabled, the Database Engine maintains versions of each row that is modified. Applications can specify that a transaction use the row versions to view data as it existed at the start of the transaction or query instead of protecting all reads with locks. By using row versioning, the chance that a read operation will block other transactions is greatly reduced.

Locking and row versioning prevent users from reading uncommitted data and prevent multiple users from attempting to change the same data at the same time. Without locking or row versioning, queries executed against that data could produce unexpected results by returning data that has not yet been committed in the database.

Applications can choose transaction isolation levels, which define the level of protection for the transaction from modifications made by other transactions. Table-level hints can be specified for individual Transact-SQL statements to further tailor behavior to fit the requirements of the application.

### Managing Concurrent Data Access

Users who access a resource at the same time are said to be accessing the resource concurrently. Concurrent data access requires mechanisms to prevent adverse effects when multiple users try to modify resources that other users are actively using.

#### Concurrency Effects

Users modifying data can affect other users who are reading or modifying the same data at the same time. These users are said to be accessing the data concurrently. If a data storage system has no concurrency control, users could see the following side effects:

* Lost updates

Lost updates occur when two or more transactions select the same row and then update the row based on the value originally selected. Each transaction is unaware of the other transactions. The last update overwrites updates made by the other transactions, which results in lost data.

For example, two editors make an electronic copy of the same document. Each editor changes the copy independently and then saves the changed copy thereby overwriting the original document. The editor who saves the changed copy last overwrites the changes made by the other editor. This problem could be avoided if one editor could not access the file until the other editor had finished and committed the transaction.

* Uncommitted dependency (dirty read)

Uncommitted dependency occurs when a second transaction selects a row that is being updated by another transaction. The second transaction is reading data that has not been committed yet and may be changed by the transaction updating the row.

For example, an editor is making changes to an electronic document. During the changes, a second editor takes a copy of the document that includes all the changes made so far, and distributes the document to the intended audience. The first editor then decides the changes made so far are wrong and removes the edits and saves the document. The distributed document contains edits that no longer exist and should be treated as if they never existed. This problem could be avoided if no one could read the changed document until the first editor does the final save of modifications and commits the transaction.

* Inconsistent analysis (nonrepeatable read)

Inconsistent analysis occurs when a second transaction accesses the same row several times and reads different data each time. Inconsistent analysis is similar to uncommitted dependency in that another transaction is changing the data that a second transaction is reading. However, in inconsistent analysis, the data read by the second transaction was committed by the transaction that made the change. Also, inconsistent analysis involves multiple reads (two or more) of the same row, and each time the information is changed by another transaction; thus, the term nonrepeatable read.

For example, an editor reads the same document twice, but between each reading the writer rewrites the document. When the editor reads the document for the second time, it has changed. The original read was not repeatable. This problem could be avoided if the writer could not change the document until the editor has finished reading it for the last time.

* Phantom reads

A phantom read is a situation that occurs when two identical queries are executed and the collection of rows returned by the second query is different. The example below shows how this may occur. Assume the two transactions below are executing at the same time. The two SELECT statements in the first transaction may return different results because the INSERT statement in the second transaction changes the data used by both.

--Transaction 1

BEGIN TRAN;

SELECT ID FROM dbo.employee

WHERE ID > 5 and ID < 10;

--The INSERT statement from the second transaction occurs here.

SELECT ID FROM dbo.employee

WHERE ID > 5 and ID < 10;

COMMIT;

--Transaction 2

BEGIN TRAN;

INSERT INTO dbo.employee

SET name = 'New' WHERE ID = 5;

COMMIT;

* Missing and double reads caused by row updates
  + Missing a updated row or seeing an updated row multiple times

Transactions that are running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. Transactions that are running at the READ COMMITTED level do issue shared locks, but the row or page locks are released after the row is read. In either case, when you are scanning an index, if another user changes the index key column of the row during your read, the row might appear again if the key change moved the row to a position ahead of your scan. Similarly, the row might not appear if the key change moved the row to a position in the index that you had already read. To avoid this, use the SERIALIZABLE or HOLDLOCK hint, or row versioning. For more information, see [Table Hints (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms187373(v=sql.110).aspx).

* + Missing one or more rows that were not the target of update

When you are using READ UNCOMMITTED, if your query reads rows using an allocation order scan (using IAM pages), you might miss rows if another transaction is causing a page split. This cannot occur when you are using read committed because a table lock is held during a page split and does not happen if the table does not have a clustered index, because updates do not cause page splits.

#### Types of Concurrency

When many people attempt to modify data in a database at the same time, a system of controls must be implemented so that modifications made by one person do not adversely affect those of another person. This is called concurrency control.

Concurrency control theory has two classifications for the methods of instituting concurrency control:

* Pessimistic concurrency control

A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

* Optimistic concurrency control

In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.

SQL Server supports a range of concurrency control. Users specify the type of concurrency control by selecting transaction isolation levels for connections or concurrency options on cursors. These attributes can be defined using Transact-SQL statements, or through the properties and attributes of database application programming interfaces (APIs) such as ADO, ADO.NET, OLE DB, and ODBC.

#### Isolation Levels in the Database Engine

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

Transaction isolation levels control:

* Whether locks are taken when data is read, and what type of locks are requested.
* How long the read locks are held.
* Whether a read operation referencing rows modified by another transaction:
  + Blocks until the exclusive lock on the row is freed.
  + Retrieves the committed version of the row that existed at the time the statement or transaction started.
  + Reads the uncommitted data modification.

|  |
| --- |
| **Important noteImportant** |
| Choosing a transaction isolation level does not affect the locks acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies, and holds that lock until the transaction completes, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions. |

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but 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 does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, may retrieve data that has been modified but not committed by other transactions. All of the concurrency side effects can happen in read uncommitted, but there is no read locking or versioning, so overhead is minimized.

#### Database Engine Isolation Levels

The ISO standard defines the following isolation levels, all of which are supported by the SQL Server Database Engine:

|  |  |
| --- | --- |
| **Isolation Level** | **Definition** |
| Read uncommitted | The lowest isolation level where transactions are isolated only enough to ensure that physically corrupt data is not read. In this level, dirty reads are allowed, so one transaction may see not-yet-committed changes made by other transactions. |
| Read committed | Allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. The Database Engine keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed. This is the Database Engine default level. |
| Repeatable read | The Database Engine keeps read and write locks that are acquired on selected data until the end of the transaction. However, because range-locks are not managed, phantom reads can occur. |
| Serializable | The highest level where transactions are completely isolated from one another. The Database Engine keeps read and write locks acquired on selected data to be released at the end of the transaction. Range-locks are acquired when a SELECT operation uses a ranged WHERE clause, especially to avoid phantom reads.   |  | | --- | | **Important noteImportant** | | DDL operations and transactions on replicated tables may fail when serializable isolation level is requested. This is because replication queries use hints that may be incompatible with serializable isolation level. | |

SQL Server also supports two additional transaction isolation levels that use row versioning. One is an implementation of read committed isolation, and one is a transaction isolation level, snapshot.

|  |  |
| --- | --- |
| **Row Versioning Isolation Level** | **Definition** |
| Read Committed Snapshot | When the READ\_COMMITTED\_SNAPSHOT database option is set ON, read committed isolation uses row versioning to provide statement-level read consistency. Read operations require only SCH-S table level locks and no page or row locks. That is, the Database Engine uses row versioning to present each statement with a transactionally consistent snapshot of the data as it existed at the start of the statement. Locks are not used to protect the data from updates by other transactions.  When the READ\_COMMITTED\_SNAPSHOT database option is set OFF, which is the default setting, read committed isolation uses shared locks to prevent other transactions from modifying rows while the current transaction is running a read operation. The shared locks also block the statement from reading rows modified by other transactions until the other transaction is completed. Both implementations meet the ISO definition of read committed isolation. |
| Snapshot | The snapshot isolation level uses row versioning to provide transaction-level read consistency. Read operations acquire no page or row locks; only SCH-S table locks are acquired. When reading rows modified by another transaction, they retrieve the version of the row that existed when the transaction started. You can only use Snapshot isolation against a database when the ALLOW\_SNAPSHOT\_ISOLATION database option is set ON. By default, this option is set OFF for user databases.   |  | | --- | | **NoteNote** | | SQL Server does not support versioning of metadata. For this reason, there are restrictions on what DDL operations can be performed in an explicit transaction that is running under snapshot isolation. The following DDL statements are not permitted under snapshot isolation after a BEGIN TRANSACTION statement: ALTER TABLE, CREATE INDEX, CREATE XML INDEX, ALTER INDEX, DROP INDEX, DBCC REINDEX, ALTER PARTITION FUNCTION, ALTER PARTITION SCHEME, or any common language runtime (CLR) DDL statement. These statements are permitted when you are using snapshot isolation within implicit transactions. An implicit transaction, by definition, is a single statement that makes it possible to enforce the semantics of snapshot isolation, even with DDL statements. Violations of this principle can cause error 3961: "Snapshot isolation transaction failed in database '%.\*ls' because the object accessed by the statement has been modified by a DDL statement in another concurrent transaction since the start of this transaction. It is not allowed because the metadata is not versioned. A concurrent update to metadata could lead to inconsistency if mixed with snapshot isolation." | |

The following table shows the concurrency side effects enabled by the different isolation levels.

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation level** | **Dirty read** | **Nonrepeatable read** | **Phantom** |
| **Read uncommitted** | Yes | Yes | Yes |
| **Read committed** | No | Yes | Yes |
| **Repeatable read** | No | No | Yes |
| **Snapshot** | No | No | No |
| **Serializable** | No | No | No |

For more information about the specific types of locking or row versioning controlled by each transaction isolation level, see [SET TRANSACTION ISOLATION LEVEL (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms173763(v=sql.110).aspx).

Transaction isolation levels can be set using Transact-SQL or through a database API.

Transact-SQL

Transact-SQL scripts use the SET TRANSACTION ISOLATION LEVEL statement.

ADO

ADO applications set the **IsolationLevel** property of the **Connection** object to adXactReadUncommitted, adXactReadCommitted, adXactRepeatableRead, or adXactReadSerializable.

ADO.NET

ADO.NET applications using the **System.Data.SqlClient** managed namespace can call the **SqlConnection.BeginTransaction** method and set the IsolationLevel option to Unspecified, Chaos, ReadUncommitted, ReadCommitted, RepeatableRead, Serializable, and Snapshot.

OLE DB

When starting a transaction, applications using OLE DB call **ITransactionLocal::StartTransaction** with isoLevel set to ISOLATIONLEVEL\_READUNCOMMITTED, ISOLATIONLEVEL\_READCOMMITTED, ISOLATIONLEVEL\_REPEATABLEREAD, ISOLATIONLEVEL\_SNAPSHOT, or ISOLATIONLEVEL\_SERIALIZABLE.

When specifying the transaction isolation level in autocommit mode, OLE DB applications can set the DBPROPSET\_SESSION property DBPROP\_SESS\_AUTOCOMMITISOLEVELS to DBPROPVAL\_TI\_CHAOS, DBPROPVAL\_TI\_READUNCOMMITTED, DBPROPVAL\_TI\_BROWSE, DBPROPVAL\_TI\_CURSORSTABILITY, DBPROPVAL\_TI\_READCOMMITTED, DBPROPVAL\_TI\_REPEATABLEREAD, DBPROPVAL\_TI\_SERIALIZABLE, DBPROPVAL\_TI\_ISOLATED, or DBPROPVAL\_TI\_SNAPSHOT.

ODBC

ODBC applications call **SQLSetConnectAttr** with Attribute set to SQL\_ATTR\_TXN\_ISOLATION and ValuePtr set to SQL\_TXN\_READ\_UNCOMMITTED, SQL\_TXN\_READ\_COMMITTED, SQL\_TXN\_REPEATABLE\_READ, or SQL\_TXN\_SERIALIZABLE.

For snapshot transactions, applications call **SQLSetConnectAttr** with Attribute set to SQL\_COPT\_SS\_TXN\_ISOLATION and ValuePtr set to SQL\_TXN\_SS\_SNAPSHOT. A snapshot transaction can be retrieved using either SQL\_COPT\_SS\_TXN\_ISOLATION or SQL\_ATTR\_TXN\_ISOLATION.

Arrow icon used with Back to Top link[[Top]](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Top)

## Locking in the Database Engine

Locking is a mechanism used by the SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time.

Before a transaction acquires a dependency on the current state of a piece of data, such as by reading or modifying the data, it must protect itself from the effects of another transaction modifying the same data. The transaction does this by requesting a lock on the piece of data. Locks have different modes, such as shared or exclusive. The lock mode defines the level of dependency the transaction has on the data. No transaction can be granted a lock that would conflict with the mode of a lock already granted on that data to another transaction. If a transaction requests a lock mode that conflicts with a lock that has already been granted on the same data, the instance of the Database Engine will pause the requesting transaction until the first lock is released.

When a transaction modifies a piece of data, it holds the lock protecting the modification until the end of the transaction. How long a transaction holds the locks acquired to protect read operations depends on the transaction isolation level setting. All locks held by a transaction are released when the transaction completes (either commits or rolls back).

Applications do not typically request locks directly. Locks are managed internally by a part of the Database Engine called the lock manager. When an instance of the Database Engine processes a Transact-SQL statement, the Database Engine query processor determines which resources are to be accessed. The query processor determines what types of locks are required to protect each resource based on the type of access and the transaction isolation level setting. The query processor then requests the appropriate locks from the lock manager. The lock manager grants the locks if there are no conflicting locks held by other transactions.

### Lock Granularity and Hierarchies

The SQL Server Database Engine has multigranular locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, the Database Engine locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions. However, it has a lower overhead because fewer locks are being maintained.

The Database Engine often has to acquire locks at multiple levels of granularity to fully protect a resource. This group of locks at multiple levels of granularity is called a lock hierarchy. For example, to fully protect a read of an index, an instance of the Database Engine may have to acquire share locks on rows and intent share locks on the pages and table.

The following table shows the resources that the Database Engine can lock.

|  |  |
| --- | --- |
| **Resource** | **Description** |
| RID | A row identifier used to lock a single row within a heap. |
| KEY | A row lock within an index used to protect key ranges in serializable transactions. |
| PAGE | An 8-kilobyte (KB) page in a database, such as data or index pages. |
| EXTENT | A contiguous group of eight pages, such as data or index pages. |
| HoBT | A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index. |
| TABLE | The entire table, including all data and indexes. |
| FILE | A database file. |
| APPLICATION | An application-specified resource. |
| METADATA | Metadata locks. |
| ALLOCATION\_UNIT | An allocation unit. |
| DATABASE | The entire database. |
| **NoteNote** | |
| HoBT and TABLE locks can be affected by the LOCK\_ESCALATION option of [ALTER TABLE](https://technet.microsoft.com/en-us/library/ms190273(v=sql.110).aspx). | |

### Lock Modes

The SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

The following table shows the resource lock modes that the Database Engine uses.

|  |  |
| --- | --- |
| **Lock mode** | **Description** |
| Shared (S) | Used for read operations that do not change or update data, such as a SELECT statement. |
| Update (U) | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| Exclusive (X) | 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). |
| 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 (BU) | 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. |

#### Shared Locks

Shared (S) locks allow concurrent transactions to read (SELECT) a resource under pessimistic concurrency control. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the read operation completes, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

#### Update Locks

Update (U) locks prevent a common form of deadlock. In a repeatable read or serializable transaction, the transaction reads data, acquiring a shared (S) lock on the resource (page or row), and then modifies the data, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.

#### Exclusive Locks

Exclusive (X) locks prevent access to a resource by concurrent transactions. With an exclusive (X) lock, no other transactions can modify data; read operations can take place only with the use of the NOLOCK hint or read uncommitted isolation level.

Data modification statements, such as INSERT, UPDATE, and DELETE combine both modification and read operations. The statement first performs read operations to acquire data before performing the required modification operations. Data modification statements, therefore, typically request both shared locks and exclusive locks. For example, an UPDATE statement might modify rows in one table based on a join with another table. In this case, the UPDATE statement requests shared locks on the rows read in the join table in addition to requesting exclusive locks on the updated rows.

#### Intent Locks

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.

For example, a shared intent lock is requested at the table level before shared (S) locks are requested on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because the Database Engine examines intent locks only at the table level to determine if a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine if a transaction can lock the entire table.

Intent locks include intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

|  |  |
| --- | --- |
| **Lock mode** | **Description** |
| Intent shared (IS) | Protects requested or acquired shared locks on some (but not all) resources lower in the hierarchy. |
| Intent exclusive (IX) | Protects requested or acquired exclusive locks on some (but not all) resources lower in the hierarchy. IX is a superset of IS, and it also protects requesting shared locks on lower level resources. |
| Shared with intent exclusive (SIX) | Protects requested or acquired shared locks on all resources lower in the hierarchy and intent exclusive locks on some (but not all) of the lower level resources. Concurrent IS locks at the top-level resource are allowed. For example, acquiring a SIX lock on a table also acquires intent exclusive locks on the pages being modified and exclusive locks on the modified rows. There can be only one SIX lock per resource at one time, preventing updates to the resource made by other transactions, although other transactions can read resources lower in the hierarchy by obtaining IS locks at the table level. |
| Intent update (IU) | Protects requested or acquired update locks on all resources lower in the hierarchy. IU locks are used only on page resources. IU locks are converted to IX locks if an update operation takes place. |
| Shared intent update (SIU) | A combination of S and IU locks, as a result of acquiring these locks separately and simultaneously holding both locks. For example, a transaction executes a query with the PAGLOCK hint and then executes an update operation. The query with the PAGLOCK hint acquires the S lock, and the update operation acquires the IU lock. |
| Update intent exclusive (UIX) | A combination of U and IX locks, as a result of acquiring these locks separately and simultaneously holding both locks. |

#### Schema Locks

The Database Engine uses schema modification (Sch-M) locks during a table data definition language (DDL) operation, such as adding a column or dropping a table. During the time that it is held, the Sch-M lock prevents concurrent access to the table. This means the Sch-M lock blocks all outside operations until the lock is released.

Some data manipulation language (DML) operations, such as table truncation, use Sch-M locks to prevent access to affected tables by concurrent operations.

The Database Engine uses schema stability (Sch-S) locks when compiling and executing queries. Sch-S locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions, including those with X locks on a table, continue to run while a query is being compiled. However, concurrent DDL operations, and concurrent DML operations that acquire Sch-M locks, cannot be performed on the table.

#### Bulk Update Locks

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. The Database Engine uses bulk update (BU) locks when both of the following conditions are true.

* You use the Transact-SQL BULK INSERT statement, or the OPENROWSET(BULK) function, or you use one of the Bulk Insert API commands such as .NET SqlBulkCopy, OLEDB Fast Load APIs, or the ODBC Bulk Copy APIs to bulk copy data into a table.
* The **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp\_tableoption**.

|  |
| --- |
| **TipTip** |
| Unlike the BULK INSERT statement, which holds a less restrictive Bulk Update lock, INSERT INTO…SELECT with the TABLOCK hint holds an exclusive (X) lock on the table. This means that you cannot insert rows using parallel insert operations. |

#### Key-Range Locks

Key-range locks protect a range of rows implicitly included in a record set being read by a Transact-SQL statement while using the serializable transaction isolation level. Key-range locking prevents phantom reads. By protecting the ranges of keys between rows, it also prevents phantom insertions or deletions into a record set accessed by a transaction.

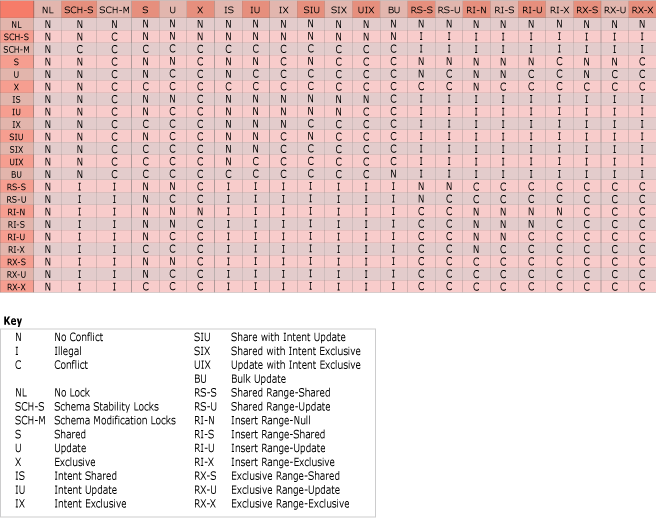
### Lock Compatibility

Lock compatibility controls whether multiple transactions can acquire locks on the same resource at the same time. If a resource is already locked by another transaction, a new lock request can be granted only if the mode of the requested lock is compatible with the mode of the existing lock. If the mode of the requested lock is not compatible with the existing lock, the transaction requesting the new lock waits for the existing lock to be released or for the lock timeout interval to expire. For example, no lock modes are compatible with exclusive locks. While an exclusive (X) lock is held, no other transaction can acquire a lock of any kind (shared, update, or exclusive) on that resource until the exclusive (X) lock is released. Alternatively, if a shared (S) lock has been applied to a resource, other transactions can also acquire a shared lock or an update (U) lock on that item even if the first transaction has not completed. However, other transactions cannot acquire an exclusive lock until the shared lock has been released.

The following table shows the compatibility of the most commonly encountered lock modes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Existing granted mode** |  |  |  |  |  |
| **Requested mode** | **IS** | **S** | **U** | **IX** | **SIX** | **X** |
| **Intent shared (IS)** | Yes | Yes | Yes | Yes | Yes | No |
| **Shared (S)** | Yes | Yes | Yes | No | No | No |
| **Update (U)** | Yes | Yes | No | No | No | No |
| **Intent exclusive (IX)** | Yes | No | No | Yes | No | No |
| **Shared with intent exclusive (SIX)** | Yes | No | No | No | No | No |
| **Exclusive (X)** | No | No | No | No | No | No |
| **NoteNote** | | | | | | | |
| An intent exclusive (IX) lock is compatible with an IX lock mode because IX means the intention is to update only some of the rows rather than all of them. Other transactions that attempt to read or update some of the rows are also permitted as long as they are not the same rows being updated by other transactions. Further, if two transactions attempt to update the same row, both transactions will be granted an IX lock at table and page level. However, one transaction will be granted an X lock at row level. The other transaction must wait until the row-level lock is removed. | | | | | | | |

Use the following table to determine the compatibility of all the lock modes available in SQL Server.



Version differences

The first in first out (FIFO) grant algorithm was significantly relaxed in SQL Server 2005 compared to SQL Server 2000. This relaxation affected requests that are compatible with all held modes and all pending modes. In these cases, the new lock could be granted immediately by passing any pending requests. Because it is compatible with all pending requests, the newly requested mode would not result in starvation. In SQL Server 2000, the new request would not be granted, because, under its stricter FIFO implementation, new requests could not be granted until all previously made requests had been granted. In the following example, connections 1 and 3 would be granted when run against SQL Server 2005 in the specified order. In SQL Server 2000, only connection 1 would be granted.

/\* Conn 1 \*/

BEGIN TRAN

EXEC sp\_getapplock 'amalgam-demo', 'IntentExclusive';

/\* Conn 2 \*/

BEGIN TRAN

EXEC sp\_getapplock 'amalgam-demo', 'Shared';

/\* Conn 3 \*/

BEGIN TRAN

EXEC sp\_getapplock 'amalgam-demo', 'IntentShare;

### Key-Range Locking

Key-range locks protect a range of rows implicitly included in a record set being read by a Transact-SQL statement while using the serializable transaction isolation level. 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.

A key-range lock is placed on an index, specifying a beginning and ending key value. This lock blocks any attempt to insert, update, or delete any row with a key value that falls in the range because those operations would first have to acquire a lock on the index. For example, a serializable transaction could issue a SELECT statement that reads all rows whose key values are between **'**AAA**'** and **'**CZZ**'**. A key-range lock on the key values in the range from **'**AAA**'** to **'**CZZ**'** prevents other transactions from inserting rows with key values anywhere in that range, such as **'**ADG**'**, **'**BBD**'**, or **'**CAL**'**.

#### Key-Range Lock Modes

Key-range locks include both a range and a row component specified in range-row format:

* Range represents the lock mode protecting the range between two consecutive index entries.
* Row represents the lock mode protecting the index entry.
* Mode represents the combined lock mode used. Key-range lock modes consist of two parts. The first represents the type of lock used to lock the index range (RangeT) and the second represents the lock type used to lock a specific key (K). The two parts are connected with a hyphen (-), such as RangeT-K.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Range** | **Row** | **Mode** | **Description** | |
| RangeS | S | RangeS-S | Shared range, shared resource lock; serializable range scan. | |
| RangeS | U | RangeS-U | Shared range, update resource lock; serializable update scan. | |
| RangeI | Null | RangeI-N | Insert range, null resource lock; used to test ranges before inserting a new key into an index. | |
| RangeX | X | RangeX-X | Exclusive range, exclusive resource lock; used when updating a key in a range. | |
| **NoteNote** | | | | |
| The internal Null lock mode is compatible with all other lock modes. | | | | |

Key-range lock modes have a compatibility matrix that shows which locks are compatible with other locks obtained on overlapping keys and ranges.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Existing granted mode** |  |  |  |  |  |  |
| **Requested mode** | **S** | **U** | **X** | **RangeS-S** | **RangeS-U** | **RangeI-N** | **RangeX-X** |
| **Shared (S)** | Yes | Yes | No | Yes | Yes | Yes | No |
| **Update (U)** | Yes | No | No | Yes | No | Yes | No |
| **Exclusive (X)** | No | No | No | No | No | Yes | No |
| **RangeS-S** | Yes | Yes | No | Yes | Yes | No | No |
| **RangeS-U** | Yes | No | No | Yes | No | No | No |
| **RangeI-N** | Yes | Yes | Yes | No | No | Yes | No |
| **RangeX-X** | No | No | No | No | No | No | No |

#### Conversion Locks

Conversion locks are created when a key-range lock overlaps another lock.

|  |  |  |
| --- | --- | --- |
| **Lock 1** | **Lock 2** | **Conversion lock** |
| S | RangeI-N | RangeI-S |
| U | RangeI-N | RangeI-U |
| X | RangeI-N | RangeI-X |
| RangeI-N | RangeS-S | RangeX-S |
| RangeI-N | RangeS-U | RangeX-U |

Conversion locks can be observed for a short period of time under different complex circumstances, sometimes while running concurrent processes.

#### Serializable Range Scan, Singleton Fetch, Delete, and Insert

Key-range locking ensures that the following operations are serializable:

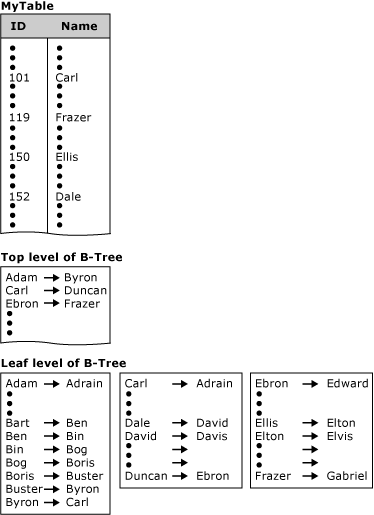
* Range scan query
* Singleton fetch of nonexistent row
* Delete operation
* Insert operation

Before key-range locking can occur, the following conditions must be satisfied:

* The transaction-isolation level must be set to SERIALIZABLE.
* The query processor must use an index to implement the range filter predicate. For example, the WHERE clause in a SELECT statement could establish a range condition with this predicate: ColumnX BETWEEN N**'**AAA**'** AND N**'**CZZ**'**. A key-range lock can only be acquired if **ColumnX** is covered by an index key.

#### Examples

The following table and index are used as a basis for the key-range locking examples that follow.



#### Range Scan Query

To ensure a range scan query is serializable, the same query should return the same results each time it is executed within the same transaction. New rows must not be inserted within the range scan query by other transactions; otherwise, these become phantom inserts. For example, the following query uses the table and index in the previous illustration:

SELECT name

FROM mytable

WHERE name BETWEEN 'A' AND 'C';

Key-range locks are placed on the index entries corresponding to the range of data rows where the name is between the values Adam and Dale, preventing new rows qualifying in the previous query from being added or deleted. Although the first name in this range is Adam, the RangeS-S mode key-range lock on this index entry ensures that no new names beginning with the letter A can be added before Adam, such as Abigail. Similarly, the RangeS-S key-range lock on the index entry for Dale ensures that no new names beginning with the letter C can be added after Carlos, such as Clive.

|  |
| --- |
| **NoteNote** |
| The number of RangeS-S locks held is n+1, where n is the number of rows that satisfy the query. |

#### Singleton Fetch of Nonexistent Data

If a query within a transaction attempts to select a row that does not exist, issuing the query at a later point within the same transaction has to return the same result. No other transaction can be allowed to insert that nonexistent row. For example, given this query:

SELECT name

FROM mytable

WHERE name = 'Bill';

A key-range lock is placed on the index entry corresponding to the name range from Ben to Bing because the name Bill would be inserted between these two adjacent index entries. The RangeS-S mode key-range lock is placed on the index entry Bing. This prevents any other transaction from inserting values, such as Bill, between the index entries Ben and Bing.

#### Delete Operation

When deleting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the delete operation. Locking the deleted key value until the end of the transaction is sufficient to maintain serializability. For example, given this DELETE statement:

DELETE mytable

WHERE name = 'Bob';

An exclusive (X) lock is placed on the index entry corresponding to the name Bob. Other transactions can insert or delete values before or after the deleted value Bob. However, any transaction that attempts to read, insert, or delete the value Bob will be blocked until the deleting transaction either commits or rolls back.

Range delete can be executed using three basic lock modes: row, page, or table lock. The row, page, or table locking strategy is decided by query optimizer or can be specified by the user through optimizer hints such as ROWLOCK, PAGLOCK, or TABLOCK. When PAGLOCK or TABLOCK is used, the Database Engine immediately deallocates an index page if all rows are deleted from this page. In contrast, when ROWLOCK is used, all deleted rows are marked only as deleted; they are removed from the index page later using a background task.

#### Insert Operation

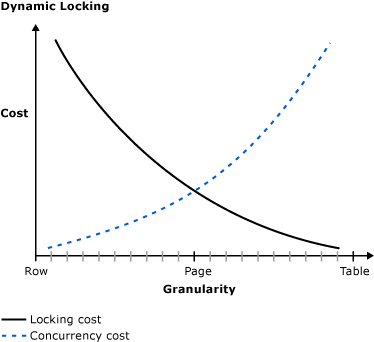
When inserting a value within a transaction, the range the value falls into does not have to be locked for the duration of the transaction performing the insert operation. Locking the inserted key value until the end of the transaction is sufficient to maintain serializability. For example, given this INSERT statement:

INSERT mytable VALUES ('Dan');

The RangeI-N mode key-range lock is placed on the index entry corresponding to the name David to test the range. If the lock is granted, Dan is inserted and an exclusive (X) lock is placed on the value Dan. The RangeI-N mode key-range lock is necessary only to test the range and is not held for the duration of the transaction performing the insert operation. Other transactions can insert or delete values before or after the inserted value Dan. However, any transaction attempting to read, insert, or delete the value Dan will be locked until the inserting transaction either commits or rolls back.

### Dynamic Locking

Using low-level locks, such as row locks, increases concurrency by decreasing the probability that two transactions will request locks on the same piece of data at the same time. Using low-level locks also increases the number of locks and the resources needed to manage them. Using high-level table or page locks lowers overhead, but at the expense of lowering concurrency.



The Microsoft SQL Server Database Engine uses a dynamic locking strategy to determine the most cost-effective locks. The Database Engine automatically determines what locks are most appropriate when the query is executed, based on the characteristics of the schema and query. For example, to reduce the overhead of locking, the optimizer may choose page-level locks in an index when performing an index scan.

Dynamic locking has the following advantages:

* Simplified database administration. Database administrators do not have to adjust lock escalation thresholds.
* Increased performance. The Database Engine minimizes system overhead by using locks appropriate to the task.
* Application developers can concentrate on development. The Database Engine adjusts locking automatically.

In SQL Server 2008 and later versions, the behavior of lock escalation has changed with the introduction of the LOCK\_ESCALATION option. For more information, see the LOCK\_ESCALATION option of [ALTER TABLE](https://technet.microsoft.com/en-us/library/ms190273(v=sql.110).aspx).

### Deadlocking

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. For example:

* Transaction A acquires a share lock on row 1.
* Transaction B acquires a share lock on row 2.
* Transaction A now requests an exclusive lock on row 2, and is blocked until transaction B finishes and releases the share lock it has on row 2.
* Transaction B now requests an exclusive lock on row 1, and is blocked until transaction A finishes and releases the share lock it has on row 1.

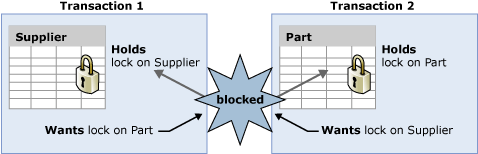
Transaction A cannot complete until transaction B completes, but transaction B is blocked by transaction A. This condition is also called a cyclic dependency: Transaction A has a dependency on transaction B, and transaction B closes the circle by having a dependency on transaction A.

Both transactions in a deadlock will wait forever unless the deadlock is broken by an external process. The SQL Server Database Engine deadlock monitor periodically checks for tasks that are in a deadlock. If the monitor detects a cyclic dependency, it chooses one of the tasks as a victim and terminates its transaction with an error. This allows the other task to complete its transaction. The application with the transaction that terminated with an error can retry the transaction, which usually completes after the other deadlocked transaction has finished.

Deadlocking is often confused with normal blocking. When a transaction requests a lock on a resource locked by another transaction, the requesting transaction waits until the lock is released. By default, SQL Server transactions do not time out, unless LOCK\_TIMEOUT is set. The requesting transaction is blocked, not deadlocked, because the requesting transaction has not done anything to block the transaction owning the lock. Eventually, the owning transaction will complete and release the lock, and then the requesting transaction will be granted the lock and proceed.

Deadlocks are sometimes called a deadly embrace.

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. For example, a thread in a multithreaded operating system might acquire one or more resources, such as blocks of memory. If the resource being acquired is currently owned by another thread, the first thread may have to wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource. In an instance of the Database Engine, sessions can deadlock when acquiring nondatabase resources, such as memory or threads.



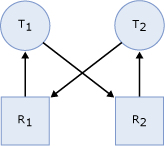
In the illustration, transaction T1 has a dependency on transaction T2 for the **Part** table lock resource. Similarly, transaction T2 has a dependency on transaction T1 for the **Supplier** table lock resource. Because these dependencies form a cycle, there is a deadlock between transactions T1 and T2.

Deadlocks can also occur when a table is partitioned and the LOCK\_ESCALATION setting of ALTER TABLE is set to AUTO. When LOCK\_ESCALATION is set to AUTO, concurrency increases by allowing the Database Engine to lock table partitions at the HoBT level instead of at the TABLE level. However, when separate transactions hold partition locks in a table and want a lock somewhere on the other transactions partition, this causes a deadlock. This type of deadlock can be avoided by setting LOCK\_ESCALATION to TABLE; although this setting will reduce concurrency by forcing large updates to a partition to wait for a table lock.

#### Detecting and Ending Deadlocks

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. The following graph presents a high level view of a deadlock state where:

* Task T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).
* Task T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to R1).
* Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.



The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock.

#### Resources That Can Deadlock

Each user session might have one or more tasks running on its behalf where each task might acquire or wait to acquire a variety of resources. The following types of resources can cause blocking that could result in a deadlock.

* Locks. Waiting to acquire locks on resources, such as objects, pages, rows, metadata, and applications can cause deadlock. For example, transaction T1 has a shared (S) lock on row r1 and is waiting to get an exclusive (X) lock on r2. Transaction T2 has a shared (S) lock on r2 and is waiting to get an exclusive (X) lock on row r1. This results in a lock cycle in which T1 and T2 wait for each other to release the locked resources.
* 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. For example, session S1 starts a transaction and acquires a shared (S) lock on row r1 and then goes to sleep. Active sessions running on all available worker threads are trying to acquire exclusive (X) locks on row r1. Because session S1 cannot acquire a worker thread, it cannot commit the transaction and release the lock on row r1. This results in a deadlock.
* Memory. When concurrent requests are waiting for memory grants that cannot be satisfied with the available memory, a deadlock can occur. For example, two concurrent queries, Q1 and Q2, execute as user-defined functions that acquire 10MB and 20MB of memory respectively. If each query needs 30MB and the total available memory is 20MB, then Q1 and Q2 must wait for each other to release memory, and this results in a deadlock.
* 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. These resources are used to control interleaving of multiple active requests under MARS. For more information, see [Multiple Active Result Sets (MARS) in SQL Server](https://msdn.microsoft.com/library/ms345109(v=sql.90).aspx).
  + 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.

In order for a task to run under MARS, it must acquire the session mutex. If the task is running under a transaction, it must then acquire the transaction mutex. This guarantees that only one task is active at one time in a given session and a given transaction. Once the required mutexes have been acquired, the task can execute. When the task finishes, or yields in the middle of the request, it will first release transaction mutex followed by the session mutex in reverse order of acquisition. However, deadlocks can occur with these resources. In the following code example, two tasks, user request U1 and user request U2, are running in the same session.

U1: Rs1=Command1.Execute("insert sometable EXEC usp\_someproc");

U2: Rs2=Command2.Execute("select colA from sometable");

The stored procedure executing from user request U1 has acquired the session mutex. If the stored procedure takes a long time to execute, it is assumed by the Database Engine that the stored procedure is waiting for input from the user. User request U2 is waiting for the session mutex while the user is waiting for the result set from U2, and U1 is waiting for a user resource. This is deadlock state logically illustrated as:

Logic diagram showing user process deadlock.

#### Deadlock Detection

All of the resources listed in the section above participate in the Database Engine deadlock detection scheme. Deadlock detection is performed by a lock monitor thread that periodically initiates a search through all of the tasks in an instance of the Database Engine. The following points describe the search process:

* The default interval is 5 seconds.
* If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks.
* If the lock monitor thread stops finding deadlocks, the Database Engine increases the intervals between searches to 5 seconds.
* If a deadlock has just been detected, it is assumed that the next threads that must wait for a lock are entering the deadlock cycle. The first couple of lock waits after a deadlock has been detected will immediately trigger a deadlock search rather than wait for the next deadlock detection interval. For example, if the current interval is 5 seconds, and a deadlock was just detected, the next lock wait will kick off the deadlock detector immediately. If this lock wait is part of a deadlock, it will be detected right away rather than during next deadlock search.

The Database Engine typically performs periodic deadlock detection only. Because the number of deadlocks encountered in the system is usually small, periodic deadlock detection helps to reduce the overhead of deadlock detection in the system.

When the lock monitor initiates deadlock search for a particular thread, it identifies the resource on which the thread is waiting. The lock monitor then finds the owner(s) for that particular resource and recursively continues the deadlock search for those threads until it finds a cycle. A cycle identified in this manner forms a deadlock.

After a deadlock is detected, the Database Engine ends a deadlock by choosing one of the threads as a deadlock victim. The Database Engine terminates the current batch being executed for the thread, rolls back the transaction of the deadlock victim, and returns a 1205 error to the application. Rolling back the transaction for the deadlock victim releases all locks held by the transaction. This allows the transactions of the other threads to become unblocked and continue. The 1205 deadlock victim error records information about the threads and resources involved in a deadlock in the error log.

By default, the Database Engine chooses as the deadlock victim the session running the transaction that is least expensive to roll back. Alternatively, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK\_PRIORITY statement. DEADLOCK\_PRIORITY can be set to LOW, NORMAL, or HIGH, or alternatively can be set to any integer value in the range (-10 to 10). The deadlock priority defaults to NORMAL. If two sessions have different deadlock priorities, the session with the lower priority is chosen as the deadlock victim. If both sessions have the same deadlock priority, the session with the transaction that is least expensive to roll back is chosen. If sessions involved in the deadlock cycle have the same deadlock priority and the same cost, a victim is chosen randomly.

When working with CLR, the deadlock monitor automatically detects deadlock for synchronization resources (monitors, reader/writer lock and thread join) accessed inside managed procedures. However, the deadlock is resolved by throwing an exception in the procedure that was selected to be the deadlock victim. It is important to understand that the exception does not automatically release resources currently owned by the victim; the resources must be explicitly released. Consistent with exception behavior, the exception used to identify a deadlock victim can be caught and dismissed.

#### Deadlock Information Tools

To view deadlock information, the Database Engine provides monitoring tools in the form of two trace flags, and the deadlock graph event in SQL Server Profiler.

#### Trace Flag 1204 and Trace Flag 1222

When deadlocks occur, trace flag 1204 and trace flag 1222 return information that is captured in the SQL Server error log. Trace flag 1204 reports deadlock information formatted by each node involved in the deadlock. Trace flag 1222 formats deadlock information, first by processes and then by resources. It is possible to enable both trace flags to obtain two representations of the same deadlock event.

In addition to defining the properties of trace flag 1204 and 1222, the following table also shows the similarities and differences.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Trace Flag 1204 and Trace Flag 1222** | **Trace Flag 1204 only** | **Trace Flag 1222 only** |
| Output format | Output is captured in the SQL Server error log. | Focused on the nodes involved in the deadlock. Each node has a dedicated section, and the final section describes the deadlock victim. | Returns information in an XML-like format that does not conform to an XML Schema Definition (XSD) schema. The format has three major sections. The first section declares the deadlock victim. The second section describes each process involved in the deadlock. The third section describes the resources that are synonymous with nodes in trace flag 1204. |
| Identifying attributes | SPID:<x> ECID:<x>. Identifies the system process ID thread in cases of parallel processes. The entry SPID:<x> ECID:0, where <x> is replaced by the SPID value, represents the main thread. The entry SPID:<x> ECID:<y>, where <x> is replaced by the SPID value and <y> is greater than 0, represents the sub-threads for the same SPID.  BatchID (sbid for trace flag 1222). Identifies the batch from which code execution is requesting or holding a lock. When Multiple Active Result Sets (MARS) is disabled, the BatchID value is 0. When MARS is enabled, the value for active batches is 1 to n. If there are no active batches in the session, BatchID is 0.  Mode. Specifies the type of lock for a particular resource that is requested, granted, or waited on by a thread. Mode can be IS (Intent Shared), S (Shared), U (Update), IX (Intent Exclusive), SIX (Shared with Intent Exclusive), and X (Exclusive).  Line # (line for trace flag 1222). Lists the line number in the current batch of statements that was being executed when the deadlock occurred.  Input Buf (inputbuf for trace flag 1222). Lists all the statements in the current batch. | Node. Represents the entry number in the deadlock chain.  Lists. The lock owner can be part of these lists:   * Grant List. Enumerates the current owners of the resource. * Convert List. Enumerates the current owners that are trying to convert their locks to a higher level. * Wait List. Enumerates current new lock requests for the resource.   Statement Type. Describes the type of DML statement (SELECT, INSERT, UPDATE, or DELETE) on which the threads have permissions.  Victim Resource Owner. Specifies the participating thread that SQL Server chooses as the victim to break the deadlock cycle. The chosen thread and all existing sub-threads are terminated.  Next Branch. Represents the two or more sub-threads from the same SPID that are involved in the deadlock cycle. | deadlock victim. Represents the physical memory address of the task (see [sys.dm\_os\_tasks (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms174963(v=sql.110).aspx)) that was selected as a deadlock victim. It may be 0 (zero) in the case of an unresolved deadlock. A task that is rolling back cannot be chosen as a deadlock victim.  executionstack. Represents Transact-SQL code that is being executed at the time the deadlock occurs.  priority. Represents deadlock priority. In certain cases, the Database Engine may opt to alter the deadlock priority for a short duration to achieve better concurrency.  logused. Log space used by the task.  owner id. The ID of the transaction that has control of the request.  status. State of the task. It is one of the following values:   * pending. Waiting for a worker thread. * runnable. Ready to run but waiting for a quantum. * running. Currently running on the scheduler. * suspended. Execution is suspended. * done. Task has completed. * spinloop. Waiting for a spinlock to become free.   waitresource. The resource needed by the task.  waittime. Time in milliseconds waiting for the resource.  schedulerid. Scheduler associated with this task. See [sys.dm\_os\_schedulers (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms177526(v=sql.110).aspx).  hostname. The name of the workstation.  isolationlevel. The current transaction isolation level.  Xactid. The ID of the transaction that has control of the request.  currentdb. The ID of the database.  lastbatchstarted. The last time a client process started batch execution.  lastbatchcompleted. The last time a client process completed batch execution.  clientoption1 and clientoption2. Set options on this client connection. This is a bitmask that includes information about options usually controlled by SET statements such as SET NOCOUNT and SET XACTABORT.  associatedObjectId. Represents the HoBT (heap or b-tree) ID. |
| Resource attributes | RID. Identifies the single row within a table on which a lock is held or requested. RID is represented as RID: db\_id:file\_id:page\_no:row\_no. For example, RID: 6:1:20789:0.  OBJECT. Identifies the table on which a lock is held or requested. OBJECT is represented as OBJECT: db\_id:object\_id. For example, TAB: 6:2009058193.  KEY. Identifies the key range within an index on which a lock is held or requested. KEY is represented as KEY: db\_id:hobt\_id (index key hash value). For example, KEY: 6:72057594057457664 (350007a4d329).  PAG. Identifies the page resource on which a lock is held or requested. PAG is represented as PAG: db\_id:file\_id:page\_no. For example, PAG: 6:1:20789.  EXT. Identifies the extent structure. EXT is represented as EXT: db\_id:file\_id:extent\_no. For example, EXT: 6:1:9.  DB. Identifies the database lock. DB is represented in one of the following ways:   * DB: db\_id * DB: db\_id[BULK-OP-DB], which identifies the database lock taken by the backup database. * DB: db\_id[BULK-OP-LOG], which identifies the lock taken by the backup log for that particular database.   APP. Identifies the lock taken by an application resource. APP is represented as APP: lock\_resource. For example, APP: Formf370f478.  METADATA. Represents metadata resources involved in a deadlock. Because METADATA has many subresources, the value returned depends upon the subresource that has deadlocked. For example, METADATA.USER\_TYPE returns user\_type\_id = <integer\_value>. For more information about METADATA resources and subresources, see [sys.dm\_tran\_locks (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms190345(v=sql.110).aspx).  HOBT. Represents a heap or b-tree involved in a deadlock. | None exclusive to this trace flag. | None exclusive to this trace flag. |

#### Trace Flag 1204 Example

The following example shows the output when trace flag 1204 is turned on. In this case, the table in Node 1 is a heap with no indexes, and the table in Node 2 is a heap with a nonclustered index. The index key in Node 2 is being updated when the deadlock occurs.

Deadlock encountered .... Printing deadlock information

Wait-for graph

Node:1

RID: 6:1:20789:0 CleanCnt:3 Mode:X Flags: 0x2

Grant List 0:

Owner:0x0315D6A0 Mode: X

Flg:0x0 Ref:0 Life:02000000 SPID:55 ECID:0 XactLockInfo: 0x04D9E27C

SPID: 55 ECID: 0 Statement Type: UPDATE Line #: 6

Input Buf: Language Event:

BEGIN TRANSACTION

   EXEC usp\_p2

Requested By:

ResType:LockOwner Stype:'OR'Xdes:0x03A3DAD0

Mode: U SPID:54 BatchID:0 ECID:0 TaskProxy:(0x04976374) Value:0x315d200 Cost:(0/868)

Node:2

KEY: 6:72057594057457664 (350007a4d329) CleanCnt:2 Mode:X Flags: 0x0

Grant List 0:

Owner:0x0315D140 Mode: X

Flg:0x0 Ref:0 Life:02000000 SPID:54 ECID:0 XactLockInfo: 0x03A3DAF4

SPID: 54 ECID: 0 Statement Type: UPDATE Line #: 6

Input Buf: Language Event:

BEGIN TRANSACTION

       EXEC usp\_p1

Requested By:

ResType:LockOwner Stype:'OR'Xdes:0x04D9E258

Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)

Victim Resource Owner:

ResType:LockOwner Stype:'OR'Xdes:0x04D9E258

Mode: U SPID:55 BatchID:0 ECID:0 TaskProxy:(0x0475E374) Value:0x315d4a0 Cost:(0/380)

#### Trace Flag 1222 Example

The following example shows the output when trace flag 1222 is turned on. In this case, one table is a heap with no indexes, and the other table is a heap with a nonclustered index. In the second table, the index key is being updated when the deadlock occurs.

deadlock-list

deadlock victim=process689978

process-list

process id=process6891f8 taskpriority=0 logused=868

waitresource=RID: 6:1:20789:0 waittime=1359 ownerId=310444

transactionname=user\_transaction

lasttranstarted=2005-09-05T11:22:42.733 XDES=0x3a3dad0

lockMode=U schedulerid=1 kpid=1952 status=suspended spid=54

sbid=0 ecid=0 priority=0 transcount=2

lastbatchstarted=2005-09-05T11:22:42.733

lastbatchcompleted=2005-09-05T11:22:42.733

clientapp=Microsoft SQL Server Management Studio - Query

hostname=TEST\_SERVER hostpid=2216 loginname=DOMAIN\user

isolationlevel=read committed (2) xactid=310444 currentdb=6

lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200

executionStack

frame procname=AdventureWorks2012.dbo.usp\_p1 line=6 stmtstart=202

sqlhandle=0x0300060013e6446b027cbb00c69600000100000000000000

UPDATE T2 SET COL1 = 3 WHERE COL1 = 1;

frame procname=adhoc line=3 stmtstart=44

sqlhandle=0x01000600856aa70f503b8104000000000000000000000000

EXEC usp\_p1

inputbuf

BEGIN TRANSACTION

       EXEC usp\_p1

process id=process689978 taskpriority=0 logused=380

waitresource=KEY: 6:72057594057457664 (350007a4d329)

waittime=5015 ownerId=310462 transactionname=user\_transaction

lasttranstarted=2005-09-05T11:22:44.077 XDES=0x4d9e258 lockMode=U

schedulerid=1 kpid=3024 status=suspended spid=55 sbid=0 ecid=0

priority=0 transcount=2 lastbatchstarted=2005-09-05T11:22:44.077

lastbatchcompleted=2005-09-05T11:22:44.077

clientapp=Microsoft SQL Server Management Studio - Query

hostname=TEST\_SERVER hostpid=2216 loginname=DOMAIN\user

isolationlevel=read committed (2) xactid=310462 currentdb=6

lockTimeout=4294967295 clientoption1=671090784 clientoption2=390200

executionStack

frame procname=AdventureWorks2012.dbo.usp\_p2 line=6 stmtstart=200

sqlhandle=0x030006004c0a396c027cbb00c69600000100000000000000

UPDATE T1 SET COL1 = 4 WHERE COL1 = 1;

frame procname=adhoc line=3 stmtstart=44

sqlhandle=0x01000600d688e709b85f8904000000000000000000000000

EXEC usp\_p2

inputbuf

BEGIN TRANSACTION

        EXEC usp\_p2

resource-list

ridlock fileid=1 pageid=20789 dbid=6 objectname=AdventureWorks2012.dbo.T2

id=lock3136940 mode=X associatedObjectId=72057594057392128

owner-list

owner id=process689978 mode=X

waiter-list

waiter id=process6891f8 mode=U requestType=wait

keylock hobtid=72057594057457664 dbid=6 objectname=AdventureWorks2012.dbo.T1

indexname=nci\_T1\_COL1 id=lock3136fc0 mode=X

associatedObjectId=72057594057457664

owner-list

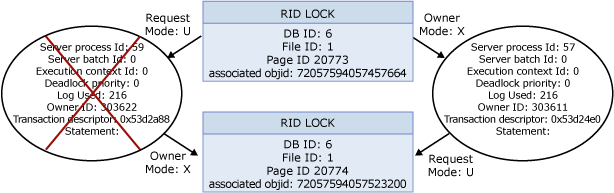
owner id=process6891f8 mode=X

waiter-list

waiter id=process689978 mode=U requestType=wait

#### Profiler Deadlock Graph Event

This is an event in SQL Server Profiler that presents a graphical depiction of the tasks and resources involved in a deadlock. The following example shows the output from SQL Server Profiler when the deadlock graph event is turned on.



For more information about running the SQL Server Profiler deadlock graph, see [Save Deadlock Graphs (SQL Server Profiler)](https://technet.microsoft.com/en-us/library/ms190465(v=sql.110).aspx).

#### Handling Deadlocks

When an instance of the SQL Server Database Engine chooses a transaction as a deadlock victim, it terminates the current batch, rolls back the transaction, and returns error message 1205 to the application.

Your transaction (process ID #52) was deadlocked on {lock | communication buffer | thread} resources with another process and has been chosen as the deadlock victim. Rerun your transaction.

Because any application submitting Transact-SQL queries can be chosen as the deadlock victim, applications should have an error handler that can trap error message 1205. If an application does not trap the error, the application can proceed unaware that its transaction has been rolled back and errors can occur.

Implementing an error handler that traps error message 1205 allows an application to handle the deadlock situation and take remedial action (for example, automatically resubmitting the query that was involved in the deadlock). By resubmitting the query automatically, the user does not need to know that a deadlock occurred.

The application should pause briefly before resubmitting its query. This gives the other transaction involved in the deadlock a chance to complete and release its locks that formed part of the deadlock cycle. This minimizes the likelihood of the deadlock reoccurring when the resubmitted query requests its locks.

#### Minimizing Deadlocks

Although deadlocks cannot be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are:

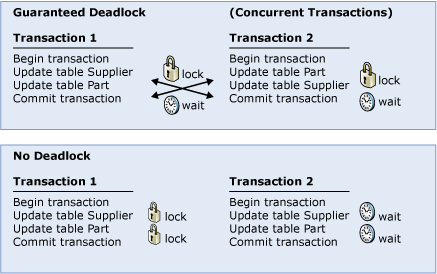
* Rolled back, undoing all the work performed by the transaction.
* Resubmitted by applications because they were rolled back when deadlocked.

To help minimize deadlocks:

* Access objects in the same order.
* Avoid user interaction in transactions.
* Keep transactions short and in one batch.
* Use a lower isolation level.
* Use a row versioning-based isolation level.
  + Set READ\_COMMITTED\_SNAPSHOT database option ON to enable read-committed transactions to use row versioning.
  + Use snapshot isolation.
* Use bound connections.

#### Access Objects in the Same Order

If all concurrent transactions access objects in the same order, deadlocks are less likely to occur. For example, if two concurrent transactions obtain a lock on the **Supplier** table and then on the **Part** table, one transaction is blocked on the **Supplier** table until the other transaction is completed. After the first transaction commits or rolls back, the second continues, and a deadlock does not occur. Using stored procedures for all data modifications can standardize the order of accessing objects.



#### Avoid User Interaction in Transactions

Avoid writing transactions that include user interaction, because the speed of batches running without user intervention is much faster than the speed at which a user must manually respond to queries, such as replying to a prompt for a parameter requested by an application. For example, if a transaction is waiting for user input and the user goes to lunch or even home for the weekend, the user delays the transaction from completing. This degrades system throughput because any locks held by the transaction are released only when the transaction is committed or rolled back. Even if a deadlock situation does not arise, other transactions accessing the same resources are blocked while waiting for the transaction to complete.

#### Keep Transactions Short and in One Batch

A deadlock typically occurs when several long-running transactions execute concurrently in the same database. The longer the transaction, the longer the exclusive or update locks are held, blocking other activity and leading to possible deadlock situations.

Keeping transactions in one batch minimizes network roundtrips during a transaction, reducing possible delays in completing the transaction and releasing locks.

#### Use a Lower Isolation Level

Determine whether a transaction can run at a lower isolation level. Implementing read committed allows a transaction to read data previously read (not modified) by another transaction without waiting for the first transaction to complete. Using a lower isolation level, such as read committed, holds shared locks for a shorter duration than a higher isolation level, such as serializable. This reduces locking contention.

#### Use a Row Versioning-based Isolation Level

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, a transaction running under read committed isolation level uses row versioning rather than shared locks during read operations.

|  |
| --- |
| **NoteNote** |
| Some applications rely upon locking and blocking behavior of read committed isolation. For these applications, some change is required before this option can be enabled. |

Snapshot isolation also uses row versioning, which does not use shared locks during read operations. Before a transaction can run under snapshot isolation, the ALLOW\_SNAPSHOT\_ISOLATION database option must be set ON.

Implement these isolation levels to minimize deadlocks that can occur between read and write operations.

#### Use Bound Connections

Using bound connections, two or more connections opened by the same application can cooperate with each other. Any locks acquired by the secondary connections are held as if they were acquired by the primary connection, and vice versa. Therefore they do not block each other.

### Lock Partitioning

For large computer systems, locks on frequently referenced objects can become a performance bottleneck as acquiring and releasing locks place contention on internal locking resources. Lock partitioning enhances locking performance by splitting a single lock resource into multiple lock resources. This feature is only available for systems with 16 or more CPUs, and is automatically enabled and cannot be disabled. Only object locks can be partitioned.Object locks that have a subtype are not partitioned. For more information, see [sys.dm\_tran\_locks (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms190345(v=sql.110).aspx).

#### Understanding Lock Partitioning

Locking tasks access several shared resources, two of which are optimized by lock partitioning:

* Spinlock. This controls access to a lock resource, such as a row or a table.

Without lock partitioning, one spinlock manages all lock requests for a single lock resource. On systems that experience a large volume of activity, contention can occur as lock requests wait for the spinlock to become available. Under this situation, acquiring locks can become a bottleneck and can negatively impact performance.

To reduce contention on a single lock resource, lock partitioning splits a single lock resource into multiple lock resources to distribute the load across multiple spinlocks.

* Memory. This is used to store the lock resource structures.

Once the spinlock is acquired, lock structures are stored in memory and then accessed and possibly modified. Distributing lock access across multiple resources helps to eliminate the need to transfer memory blocks between CPUs, which will help to improve performance.

#### Implementing and Monitoring Lock Partitioning

Lock partitioning is turned on by default for systems with 16 or more CPUs. When lock partitioning is enabled, an informational message is recorded in the SQL Server error log.

When acquiring locks on a partitioned resource:

* Only NL, SCH-S, IS, IU, and IX lock modes are acquired on a single partition.
* Shared (S), exclusive (X), and other locks in modes other than NL, SCH-S, IS, IU, and IX must be acquired on all partitions starting with partition ID 0 and following in partition ID order. These locks on a partitioned resource will use more memory than locks in the same mode on a non-partitioned resource since each partition is effectively a separate lock. The memory increase is determined by the number of partitions. The SQL Server lock counters in the Windows Performance Monitor will display information about memory used by partitioned and non-partitioned locks.

A transaction is assigned to a partition when the transaction starts. For the transaction, all lock requests that can be partitioned use the partition assigned to that transaction. By this method, access to lock resources of the same object by different transactions is distributed across different partitions.

The resource\_lock\_partition column in the sys.dm\_tran\_locks Dynamic Management View provides the lock partition ID for a lock partitioned resource. For more information, see [sys.dm\_tran\_locks (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms190345(v=sql.110).aspx).

Under the Locks event in SQL Server Profiler, the BigintData1 column provides the lock partition ID for a lock partitioned resource.

#### Working with Lock Partitioning

The following code examples illustrate lock partitioning. In the examples, two transactions are executed in two different sessions in order to show lock partitioning behavior on a computer system with 16 CPUs.

These Transact-SQL statements create test objects that are used in the examples that follow.

-- Create a test table.

CREATE TABLE TestTable

(col1 int);

GO

-- Create a clustered index on the table.

CREATE CLUSTERED INDEX ci\_TestTable

ON TestTable (col1);

GO

-- Populate the table.

INSERT INTO TestTable VALUES (1);

GO

#### Example A

Session 1:

A SELECT statement is executed under a transaction. Because of the HOLDLOCK lock hint, this statement will acquire and retain an Intent shared (IS) lock on the table (for this illustration, row and page locks are ignored). The IS lock will be acquired only on the partition assigned to the transaction. For this example, it is assumed that the IS lock is acquired on partition ID 7.

-- Start a transaction.

BEGIN TRANSACTION

-- This SELECT statement will acquire an IS lock on the table.

SELECT col1

FROM TestTable

WITH (HOLDLOCK);

Session 2:

A transaction is started, and the SELECT statement running under this transaction will acquire and retain a shared (S) lock on the table. The S lock will be acquired on all partitions which results in multiple table locks, one for each partition. For example, on a 16-cpu system, 16 S locks will be issued across lock partition IDs 0-15. Because the S lock is compatible with the IS lock being held on partition ID 7 by the transaction in session 1, there is no blocking between transactions.

BEGIN TRANSACTION

SELECT col1

FROM TestTable

WITH (TABLOCK, HOLDLOCK);

Session 1:

The following SELECT statement is executed under the transaction that is still active under session 1. Because of the exclusive (X) table lock hint, the transaction will attempt to acquire an X lock on the table. However, the S lock that is being held by the transaction in session 2 will block the X lock at partition ID 0.

SELECT col1

FROM TestTable

WITH (TABLOCKX);

#### Example B

Session 1:

A SELECT statement is executed under a transaction. Because of the HOLDLOCK lock hint, this statement will acquire and retain an Intent shared (IS) lock on the table (for this illustration, row and page locks are ignored). The IS lock will be acquired only on the partition assigned to the transaction. For this example, it is assumed that the IS lock is acquired on partition ID 6.

-- Start a transaction.

BEGIN TRANSACTION

-- This SELECT statement will acquire an IS lock on the table.

SELECT col1

FROM TestTable

WITH (HOLDLOCK);

Session 2:

A SELECT statement is executed under a transaction. Because of the TABLOCKX lock hint, the transaction tries to acquire an exclusive (X) lock on the table. Remember that the X lock must be acquired on all partitions starting with partition ID 0. The X lock will be acquired on all partitions IDs 0-5 but will be blocked by the IS lock that is acquired on partition ID 6.

On partition IDs 7-15 that the X lock has not yet reached, other transactions can continue to acquire locks.

BEGIN TRANSACTION

SELECT col1

FROM TestTable

WITH (TABLOCKX, HOLDLOCK);

Arrow icon used with Back to Top link[[Top]](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Top)

## Row Versioning-based Isolation Levels in the Database Engine

Starting with SQL Server 2005, the Database Engine offers an implementation of an existing transaction isolation level, read committed, that provides a statement level snapshot using row versioning. SQL Server Database Engine also offers a transaction isolation level, snapshot, that provides a transaction level snapshot also using row versioning.

Row versioning is a general framework in SQL Server that invokes a copy-on-write mechanism when a row is modified or deleted. This requires that while the transaction is running, the old version of the row must be available for transactions that require an earlier transactionally consistent state. Row versioning is used to do the following:

* Build the **inserted** and **deleted** tables in triggers. Any rows modified by the trigger are versioned. This includes the rows modified by the statement that launched the trigger, as well as any data modifications made by the trigger.
* Support Multiple Active Result Sets (MARS). If a MARS session issues a data modification statement (such as INSERT, UPDATE, or DELETE) at a time there is an active result set, the rows affected by the modification statement are versioned.
* Support index operations that specify the ONLINE option.
* Support row versioning-based transaction isolation levels:
  + A new implementation of read committed isolation level that uses row versioning to provide statement-level read consistency.
  + A new isolation level, snapshot, to provide transaction-level read consistency.

The **tempdb** database must have enough space for the version store. When **tempdb** is full, update operations will stop generating versions and continue to succeed, but read operations might fail because a particular row version that is needed no longer exists. This affects operations like triggers, MARS, and online indexing.

Using row versioning for read-committed and snapshot transactions is a two-step process:

1. Set either or both the READ\_COMMITTED\_SNAPSHOT and ALLOW\_SNAPSHOT\_ISOLATION database options ON.
2. Set the appropriate transaction isolation level in an application:
   * When the READ\_COMMITTED\_SNAPSHOT database option is ON, transactions setting the read committed isolation level use row versioning.
   * When the ALLOW\_SNAPSHOT\_ISOLATION database option is ON, transactions can set the snapshot isolation level.

When either READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database option is set ON, the SQL Server Database Engine assigns a transaction sequence number (XSN) to each transaction that manipulates data using row versioning. Transactions start at the time a BEGIN TRANSACTION statement is executed. However, the transaction sequence number starts with the first read or write operation after the BEGIN TRANSACTION statement. The transaction sequence number is incremented by one each time it is assigned.

When either the READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION database options are ON, logical copies (versions) are maintained for all data modifications performed in the database. Every time a row is modified by a specific transaction, the instance of the Database Engine stores a version of the previously committed image of the row in **tempdb**. Each version is marked with the transaction sequence number of the transaction that made the change. The versions of modified rows are chained using a link list. The newest row value is always stored in the current database and chained to the versioned rows stored in **tempdb**.

|  |
| --- |
| **NoteNote** |
| For modification of large objects (LOBs), only the changed fragment is copied to the version store in **tempdb**. |

Row versions are held long enough to satisfy the requirements of transactions running under row versioning-based isolation levels. The Database Engine tracks the earliest useful transaction sequence number and periodically deletes all row versions stamped with transaction sequence numbers that are lower than the earliest useful sequence number.

When both database options are set to OFF, only rows modified by triggers or MARS sessions, or read by ONLINE index operations, are versioned. Those row versions are released when no longer needed. A background thread periodically executes to remove stale row versions.

|  |
| --- |
| **NoteNote** |
| For short-running transactions, a version of a modified row may get cached in the buffer pool without getting written into the disk files of the **tempdb** database. If the need for the versioned row is short-lived, it will simply get dropped from the buffer pool and may not necessarily incur I/O overhead. |

### Behavior When Reading Data

When transactions running under row versioning-based isolation read data, the read operations do not acquire shared (S) locks on the data being read, and therefore do not block transactions that are modifying data. Also, the overhead of locking resources is minimized as the number of locks acquired is reduced. Read committed isolation using row versioning and snapshot isolation are designed to provide statement-level or transaction-level read consistencies of versioned data.

All queries, including transactions running under row versioning-based isolation levels, acquire Sch-S (schema stability) locks during compilation and execution. Because of this, queries are blocked when a concurrent transaction holds a Sch-M (schema modification) lock on the table. For example, a data definition language (DDL) operation acquires a Sch-M lock before it modifies the schema information of the table. Query transactions, including those running under a row versioning-based isolation level, are blocked when attempting to acquire a Sch-S lock. Conversely, a query holding a Sch-S lock blocks a concurrent transaction that attempts to acquire a Sch-M lock.

When a transaction using the snapshot isolation level starts, the instance of the Database Engine records all of the currently active transactions. When the snapshot transaction reads a row that has a version chain, the Database Engine follows the chain and retrieves the row where the transaction sequence number is:

* Closest to but lower than the sequence number of the snapshot transaction reading the row.
* Not in the list of the transactions active when the snapshot transaction started.

Read operations performed by a snapshot transaction retrieve the last version of each row that had been committed at the time the snapshot transaction started. This provides a transactionally consistent snapshot of the data as it existed at the start of the transaction.

Read-committed transactions using row versioning operate in much the same way. The difference is that the read-committed transaction does not use its own transaction sequence number when choosing row versions. Each time a statement is started, the read-committed transaction reads the latest transaction sequence number issued for that instance of the Database Engine. This is the transaction sequence number used to select the correct row versions for that statement. This allows read-committed transactions to see a snapshot of the data as it exists at the start of each statement.

|  |
| --- |
| **NoteNote** |
| Even though read-committed transactions using row versioning provides a transactionally consistent view of the data at a statement level, row versions generated or accessed by this type of transaction are maintained until the transaction completes. |

### Behavior When Modifying Data

In a read-committed transaction using row versioning, the selection of rows to update is done using a blocking scan where an update (U) lock is taken on the data row as data values are read. This is the same as a read-committed transaction that does not use row versioning. If the data row does not meet the update criteria, the update lock is released on that row and the next row is locked and scanned.

Transactions running under snapshot isolation take an optimistic approach to data modification by acquiring locks on data before performing the modification only to enforce constraints. Otherwise, locks are not acquired on data until the data is to be modified. When a data row meets the update criteria, the snapshot transaction verifies that the data row has not been modified by a concurrent transaction that committed after the snapshot transaction began. If the data row has been modified outside of the snapshot transaction, an update conflict occurs and the snapshot transaction is terminated. The update conflict is handled by the Database Engine and there is no way to disable the update conflict detection.

|  |
| --- |
| **NoteNote** |
| Update operations running under snapshot isolation internally execute under read committed isolation when the snapshot transaction accesses any of the following:  A table with a FOREIGN KEY constraint.  A table that is referenced in the FOREIGN KEY constraint of another table.  An indexed view referencing more than one table.  However, even under these conditions the update operation will continue to verify that the data has not been modified by another transaction. If data has been modified by another transaction, the snapshot transaction encounters an update conflict and is terminated. |

### Behavior in Summary

The following table summarizes the differences between snapshot isolation and read committed isolation using row versioning.

|  |  |  |
| --- | --- | --- |
| **Property** | **Read-committed isolation level using row versioning** | **Snapshot isolation level** |
| The database option that must be set to ON to enable the required support. | READ\_COMMITTED\_SNAPSHOT | ALLOW\_SNAPSHOT\_ISOLATION |
| How a session requests the specific type of row versioning. | Use the default read-committed isolation level, or run the SET TRANSACTION ISOLATION LEVEL statement to specify the READ COMMITTED isolation level. This can be done after the transaction starts. | Requires the execution of SET TRANSACTION ISOLATION LEVEL to specify the SNAPSHOT isolation level before the start of the transaction. |
| The version of data read by statements. | All data that was committed before the start of each statement. | All data that was committed before the start of each transaction. |
| How updates are handled. | Reverts from row versions to actual data to select rows to update and uses update locks on the data rows selected. Acquires exclusive locks on actual data rows to be modified. No update conflict detection. | Uses row versions to select rows to update. Tries to acquire an exclusive lock on the actual data row to be modified, and if the data has been modified by another transaction, an update conflict occurs and the snapshot transaction is terminated. |
| Update conflict detection. | None. | Integrated support. Cannot be disabled. |

### Row Versioning Resource Usage

The row versioning framework supports the following features available in SQL Server:

* Triggers
* Multiple Active Results Sets (MARS)
* Online indexing

The row versioning framework also supports the following row versioning-based transaction isolation levels, which by default are not enabled:

* When the READ\_COMMITTED\_SNAPSHOT database option is ON, READ\_COMMITTED transactions provide statement-level read consistency using row versioning.
* When the ALLOW\_SNAPSHOT\_ISOLATION database option is ON, SNAPSHOT transactions provide transaction-level read consistency using row versioning.

Row versioning-based isolation levels reduce the number of locks acquired by transaction by eliminating the use of shared locks on read operations. This increases system performance by reducing the resources used to manage locks. Performance is also increased by reducing the number of times a transaction is blocked by locks acquired by other transactions.

Row versioning-based isolation levels increase the resources needed by data modifications. Enabling these options causes all data modifications for the database to be versioned. A copy of the data before modification is stored in tempdb even when there are no active transactions using row versioning-based isolation. The data after modification includes a pointer to the versioned data stored in tempdb. For large objects, only part of the object that changed is copied to tempdb.

#### Space Used in tempdb

For each instance of the Database Engine, tempdb must have enough space to hold the row versions generated for every database in the instance. The database administrator must ensure that tempdb has ample space to support the version store. There are two version stores in tempdb:

* The online index build version store is used for online index builds in all databases.
* The common version store is used for all other data modification operations in all databases.

Row versions must be stored for as long as an active transaction needs to access it. Once every minute, a background thread removes row versions that are no longer needed and frees up the version space in tempdb. A long-running transaction prevents space in the version store from being released if it meets any of the following conditions:

* It uses row versioning-based isolation.
* It uses triggers, MARS, or online index build operations.
* It generates row versions.

|  |
| --- |
| **NoteNote** |
| When a trigger is invoked inside a transaction, the row versions created by the trigger are maintained until the end of the transaction, even though the row versions are no longer needed after the trigger completes. This also applies to read-committed transactions that use row versioning. With this type of transaction, a transactionally consistent view of the database is needed only for each statement in the transaction. This means that the row versions created for a statement in the transaction are no longer needed after the statement completes. However, row versions created by each statement in the transaction are maintained until the transaction completes. |

When tempdb runs out of space, the Database Engine forces the version stores to shrink. During the shrink process, the longest running transactions that have not yet generated row versions are marked as victims. A message 3967 is generated in the error log for each victim transaction. If a transaction is marked as a victim, it can no longer read the row versions in the version store. When it attempts to read row versions, message 3966 is generated and the transaction is rolled back. If the shrinking process succeeds, space becomes available in tempdb. Otherwise, tempdb runs out of space and the following occurs:

* Write operations continue to execute but do not generate versions. An information message (3959) appears in the error log, but the transaction that writes data is not affected.
* Transactions that attempt to access row versions that were not generated because of a tempdb full rollback terminate with an error 3958.

#### Space Used in Data Rows

Each database row may use up to 14 bytes at the end of the row for row versioning information. The row versioning information contains the transaction sequence number of the transaction that committed the version and the pointer to the versioned row. These 14 bytes are added the first time the row is modified, or when a new row is inserted, under any of these conditions:

* READ\_COMMITTED\_SNAPSHOT or ALLOW\_SNAPSHOT\_ISOLATION options are ON.
* The table has a trigger.
* Multiple Active Results Sets (MARS) is being used.
* Online index build operations are currently running on the table.

These 14 bytes are removed from the database row the first time the row is modified under all of these conditions:

* READ\_COMMITTED\_SNAPSHOT and ALLOW\_SNAPSHOT\_ISOLATION options are OFF.
* The trigger no longer exists on the table.
* MARS is not being used.
* Online index build operations are not currently running.

If you use any of the row versioning features, you might need to allocate additional disk space for the database to accommodate the 14 bytes per database row. Adding the row versioning information can cause index page splits or the allocation of a new data page if there is not enough space available on the current page. For example, if the average row length is 100 bytes, the additional 14 bytes cause an existing table to grow up to 14 percent.

Decreasing the [fill factor](https://technet.microsoft.com/en-us/library/ms177459(v=sql.110).aspx) might help to prevent or decrease fragmentation of index pages. To view fragmentation information for the data and indexes of a table or view, you can use [DBCC SHOWCONTIG](https://technet.microsoft.com/en-us/library/ms175008(v=sql.110).aspx).

#### Space Used in Large Objects

The SQL Server Database Engine supports six data types that can hold large strings up to 2 gigabytes (GB) in length: nvarchar(max), varchar(max), varbinary(max), ntext, text, and image. Large strings stored using these data types are stored in a series of data fragments that are linked to the data row. Row versioning information is stored in each fragment used to store these large strings. Data fragments are a collection of pages dedicated to large objects in a table.

As new large values are added to a database, they are allocated using a maximum of 8040 bytes of data per fragment. Earlier versions of the Database Engine stored up to 8080 bytes of ntext, text, or image data per fragment.

Existing ntext, text, and image large object (LOB) data is not updated to make space for the row versioning information when a database is upgraded to SQL Server from an earlier version of SQL Server. However, the first time the LOB data is modified, it is dynamically upgraded to enable storage of versioning information. This will happen even if row versions are not generated. After the LOB data is upgraded, the maximum number of bytes stored per fragment is reduced from 8080 bytes to 8040 bytes. The upgrade process is equivalent to deleting the LOB value and reinserting the same value. The LOB data is upgraded even if only one byte is modified. This is a one-time operation for each ntext, text, or image column, but each operation may generate a large amount of page allocations and I/O activity depending upon the size of the LOB data. It may also generate a large amount of logging activity if the modification is fully logged. WRITETEXT and UPDATETEXT operations are minimally logged if database recovery mode is not set to FULL.

The nvarchar(max), varchar(max), and varbinary(max) data types are not available in earlier versions of SQL Server. Therefore, they have no upgrade issues.

Enough disk space should be allocated to accommodate this requirement.

#### Monitoring Row Versioning and the Version Store

For monitoring row versioning, version store, and snapshot isolation processes for performance and problems, SQL Server provides tools in the form of Dynamic Management Views (DMVs) and performance counters in Windows System Monitor.

#### DMVs

The following DMVs provide information about the current system state of tempdb and the version store, as well as transactions using row versioning.

sys.dm\_db\_file\_space\_usage. Returns space usage information for each file in the database. For more information, see [sys.dm\_db\_file\_space\_usage (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms174412(v=sql.110).aspx).

sys.dm\_db\_session\_space\_usage. Returns page allocation and deallocation activity by session for the database. For more information, see [sys.dm\_db\_session\_space\_usage (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms187938(v=sql.110).aspx).

sys.dm\_db\_task\_space\_usage. Returns page allocation and deallocation activity by task for the database. For more information, see [sys.dm\_db\_task\_space\_usage (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms190288(v=sql.110).aspx).

sys.dm\_tran\_top\_version\_generators. Returns a virtual table for the objects producing the most versions in the version store. It groups the top 256 aggregated record lengths by database\_id and rowset\_id. Use this function to find the largest consumers of the version store. For more information, see [sys.dm\_tran\_top\_version\_generators (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms188778(v=sql.110).aspx).

sys.dm\_tran\_version\_store. Returns a virtual table that displays all version records in the common version store. For more information, see [sys.dm\_tran\_version\_store (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms186328(v=sql.110).aspx).

|  |
| --- |
| **NoteNote** |
| sys.dm\_tran\_top\_version\_generators and sys.dm\_tran\_version\_store are potentially very expensive functions to run, since both query the entire version store, which could be very large. |

sys.dm\_tran\_active\_snapshot\_database\_transactions. Returns a virtual table for all active transactions in all databases within the SQL Server instance that use row versioning. System transactions do not appear in this DMV. For more information, see [sys.dm\_tran\_active\_snapshot\_database\_transactions (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms180023(v=sql.110).aspx).

sys.dm\_tran\_transactions\_snapshot. Returns a virtual table that displays snapshots taken by each transaction. The snapshot contains the sequence number of the active transactions that use row versioning. For more information, see [sys.dm\_tran\_transactions\_snapshot (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms173540(v=sql.110).aspx).

sys.dm\_tran\_current\_transaction. Returns a single row that displays row versioning-related state information of the transaction in the current session. For more information, see [sys.dm\_tran\_current\_transaction (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms186327(v=sql.110).aspx).

sys.dm\_tran\_current\_snapshot. Returns a virtual table that displays all active transactions at the time the current snapshot isolation transaction starts. If the current transaction is using snapshot isolation, this function returns no rows. sys.dm\_tran\_current\_snapshot is similar to sys.dm\_tran\_transactions\_snapshot, except that it returns only the active transactions for the current snapshot. For more information, see [sys.dm\_tran\_current\_snapshot (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms184390(v=sql.110).aspx).

#### Performance Counters

SQL Server performance counters provide information about the system performance impacted by SQL Server processes. The following performance counters monitor tempdb and the version store, as well as transactions using row versioning. The performance counters are contained in the SQLServer:Transactions performance object.

Free Space in tempdb (KB). Monitors the amount, in kilobytes (KB), of free space in the tempdb database. There must be enough free space in tempdb to handle the version store that supports snapshot isolation.

The following formula provides a rough estimate of the size of the version store. For long-running transactions, it may be useful to monitor the generation and cleanup rate to estimate the maximum size of the version store.

[size of common version store] = 2 \* [version store data generated per minute] \* [longest running time (minutes) of the transaction]

The longest running time of transactions should not include online index builds. Because these operations may take a long time on very large tables, online index builds use a separate version store. The approximate size of the online index build version store equals the amount of data modified in the table, including all indexes, while the online index build is active.

Version Store Size (KB). Monitors the size in KB of all version stores. This information helps determine the amount of space needed in the tempdb database for the version store. Monitoring this counter over a period of time provides a useful estimate of additional space needed for tempdb.

Version Generation rate (KB/s). Monitors the version generation rate in KB per second in all version stores.

Version Cleanup rate (KB/s). Monitors the version cleanup rate in KB per second in all version stores.

|  |
| --- |
| **NoteNote** |
| Information from Version Generation rate (KB/s) and Version Cleanup rate (KB/s) can be used to predict tempdb space requirements. |

Version Store unit count. Monitors the count of version store units.

Version Store unit creation. Monitors the total number of version store units created to store row versions since the instance was started.

Version Store unit truncation. Monitors the total number of version store units truncated since the instance was started. A version store unit is truncated when SQL Server determines that none of the version rows stored in the version store unit are needed to run active transactions.

Update conflict ratio. Monitors the ratio of update snapshot transaction that have update conflicts to the total number of update snapshot transactions.

Longest Transaction Running Time. Monitors the longest running time in seconds of any transaction using row versioning. This can be used to determine if any transaction is running for an unreasonable amount of time.

Transactions. Monitors the total number of active transactions. This does not include system transactions.

Snapshot Transactions. Monitors the total number of active snapshot transactions.

Update Snapshot Transactions. Monitors the total number of active snapshot transactions that perform update operations.

NonSnapshot Version Transactions. Monitors the total number of active non-snapshot transactions that generate version records.

|  |
| --- |
| **NoteNote** |
| The sum of Update Snapshot Transactions and NonSnapshot Version Transactions represents the total number of transactions that participate in version generation. The difference of Snapshot Transactions and Update Snapshot Transactions reports the number of read-only snapshot transactions. |

### Row Versioning-based Isolation Level Example

The following examples show the differences in behavior between snapshot isolation transactions and read-committed transactions that use row versioning.

#### A. Working with snapshot isolation

In this example, a transaction running under snapshot isolation reads data that is then modified by another transaction. The snapshot transaction does not block the update operation executed by the other transaction, and it continues to read data from the versioned row, ignoring the data modification. However, when the snapshot transaction attempts to modify the data that has already been modified by the other transaction, the snapshot transaction generates an error and is terminated.

On session 1:

USE AdventureWorks2012; -- Or the 2008 or 2008R2 version of the AdventureWorks database.

GO

-- Enable snapshot isolation on the database.

ALTER DATABASE AdventureWorks2012

SET ALLOW\_SNAPSHOT\_ISOLATION ON;

GO

-- Start a snapshot transaction

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

GO

BEGIN TRANSACTION;

-- This SELECT statement will return

-- 48 vacation hours for the employee.

SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 2:

USE AdventureWorks2012;

GO

-- Start a transaction.

BEGIN TRANSACTION;

-- Subtract a vacation day from employee 4.

-- Update is not blocked by session 1 since

-- under snapshot isolation shared locks are

-- not requested.

UPDATE HumanResources.Employee

SET VacationHours = VacationHours - 8

WHERE BusinessEntityID = 4;

    -- Verify that the employee now has 40 vacation hours.

SELECT VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 1:

-- Reissue the SELECT statement - this shows

-- the employee having 48 vacation hours. The

-- snapshot transaction is still reading data from

-- the versioned row.

    SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 2:

-- Commit the transaction; this commits the data

-- modification.

COMMIT TRANSACTION;

GO

On session 1:

-- Reissue the SELECT statement - this still

-- shows the employee having 48 vacation hours

-- even after the other transaction has committed

-- the data modification.

    SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

-- Because the data has been modified outside of the

-- snapshot transaction, any further data changes to

-- that data by the snapshot transaction will cause

-- the snapshot transaction to fail. This statement

-- will generate a 3960 error and the transaction will

-- terminate.

UPDATE HumanResources.Employee

SET SickLeaveHours = SickLeaveHours - 8

WHERE BusinessEntityID = 4;

-- Undo the changes to the database from session 1.

-- This will not undo the change from session 2.

ROLLBACK TRANSACTION

GO

#### B. Working with read-committed using row versioning

In this example, a read-committed transaction using row versioning runs concurrently with another transaction. The read-committed transaction behaves differently than a snapshot transaction. Like a snapshot transaction, the read-committed transaction will read versioned rows even after the other transaction has modified data. However, unlike a snapshot transaction, the read-committed transaction will:

* Read the modified data after the other transaction commits the data changes.
* Be able to update the data modified by the other transaction where the snapshot transaction could not.

On session 1:

USE AdventureWorks2012; -- Or any earlier version of the AdventureWorks database.

GO

-- Enable READ\_COMMITTED\_SNAPSHOT on the database.

-- For this statement to succeed, this session

-- must be the only connection to the AdventureWorks2012

-- database.

ALTER DATABASE AdventureWorks2012

SET READ\_COMMITTED\_SNAPSHOT ON;

GO

-- Start a read-committed transaction

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

GO

BEGIN TRANSACTION;

-- This SELECT statement will return

-- 48 vacation hours for the employee.

SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 2:

USE AdventureWorks2012;

GO

-- Start a transaction.

BEGIN TRANSACTION;

-- Subtract a vacation day from employee 4.

-- Update is not blocked by session 1 since

-- under read-committed using row versioning shared locks are

-- not requested.

UPDATE HumanResources.Employee

SET VacationHours = VacationHours - 8

WHERE BusinessEntityID = 4;

    -- Verify that the employee now has 40 vacation hours.

SELECT VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 1:

-- Reissue the SELECT statement - this still shows

-- the employee having 48 vacation hours. The

-- read-committed transaction is still reading data

-- from the versioned row and the other transaction

-- has not committed the data changes yet.

    SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

On session 2:

-- Commit the transaction.

COMMIT TRANSACTION;

GO

On session 1:

-- Reissue the SELECT statement which now shows the

-- employee having 40 vacation hours. Being

-- read-committed, this transaction is reading the

-- committed data. This is different from snapshot

-- isolation which reads from the versioned row.

SELECT BusinessEntityID, VacationHours

FROM HumanResources.Employee

WHERE BusinessEntityID = 4;

-- This statement, which caused the snapshot transaction

-- to fail, will succeed with read-committed using row versioning.

UPDATE HumanResources.Employee

SET SickLeaveHours = SickLeaveHours - 8

WHERE BusinessEntityID = 4;

-- Undo the changes to the database from session 1.

-- This will not undo the change from session 2.

ROLLBACK TRANSACTION;

GO

### Enabling Row Versioning-Based Isolation Levels

Database administrators control the database-level settings for row versioning by using the READ\_COMMITTED\_SNAPSHOT and ALLOW\_SNAPSHOT\_ISOLATION database options in the ALTER DATABASE statement.

When the READ\_COMMITTED\_SNAPSHOT database option is set ON, the mechanisms used to support the option are activated immediately. When setting the READ\_COMMITTED\_SNAPSHOT option, only the connection executing the ALTER DATABASE command is allowed in the database. There must be no other open connection in the database until ALTER DATABASE is complete. The database does not have to be in single-user mode.

The following Transact-SQL statement enables READ\_COMMITTED\_SNAPSHOT:

ALTER DATABASE AdventureWorks2012

SET READ\_COMMITTED\_SNAPSHOT ON;

When the ALLOW\_SNAPSHOT\_ISOLATION database option is set ON, the instance of the SQL Server Database Engine does not generate row versions for modified data until all active transactions that have modified data in the database complete. If there are active modification transactions, SQL Server sets the state of the option to PENDING\_ON. After all of the modification transactions complete, the state of the option is changed to ON. Users cannot start a snapshot transaction in that database until the option is fully ON. The database passes through a PENDING\_OFF state when the database administrator sets the ALLOW\_SNAPSHOT\_ISOLATION option to OFF.

The following Transact-SQL statement will enable ALLOW\_SNAPSHOT\_ISOLATION:

ALTER DATABASE AdventureWorks2012

SET ALLOW\_SNAPSHOT\_ISOLATION ON;

The following table lists and describes the states of the ALLOW\_SNAPSHOT\_ISOLATION option. Using ALTER DATABASE with the ALLOW\_SNAPSHOT\_ISOLATION option does not block users who are currently accessing the database data.

|  |  |
| --- | --- |
| **State of snapshot isolation framework for current database** | **Description** |
| OFF | The support for snapshot isolation transactions is not activated. No snapshot isolation transactions are allowed. |
| PENDING\_ON | The support for snapshot isolation transactions is in transition state (from OFF to ON). Open transactions must complete.  No snapshot isolation transactions are allowed. |
| ON | The support for snapshot isolation transactions is activated.  Snapshot transactions are allowed. |
| PENDING\_OFF | The support for snapshot isolation transactions is in transition state (from ON to OFF).  Snapshot transactions started after this time cannot access this database. Update transactions still pay the cost of versioning in this database. Existing snapshot transactions can still access this database without a problem. The state PENDING\_OFF does not become OFF until all snapshot transactions that were active when the database snapshot isolation state was ON finish. |

Use the sys.databases catalog view to determine the state of both row versioning database options.

All updates to user tables and some system tables stored in master and msdb generate row versions.

The ALLOW\_SNAPSHOT\_ISOLATION option is automatically set ON in the master and msdb databases, and cannot be disabled.

Users cannot set the READ\_COMMITTED\_SNAPSHOT option ON in master, tempdb, or msdb.

### Using Row Versioning-based Isolation Levels

The row versioning framework is always enabled in SQL Server, and is used by multiple features. Besides providing row versioning-based isolation levels, it is used to support modifications made in triggers and multiple active result sets (MARS) sessions, and to support data reads for ONLINE index operations.

Row versioning-based isolation levels are enabled at the database level. Any application accessing objects from enabled databases can run queries using the following isolation levels:

* Read-committed that uses row versioning by setting the READ\_COMMITTED\_SNAPSHOT database option to ON as shown in the following code example:
* ALTER DATABASE AdventureWorks2012
* SET READ\_COMMITTED\_SNAPSHOT ON;

When the database is enabled for READ\_COMMITTED\_SNAPSHOT, all queries running under the read committed isolation level use row versioning, which means that read operations do not block update operations.

* Snapshot isolation by setting the ALLOW\_SNAPSHOT\_ISOLATION database option to ON as shown in the following code example:
* ALTER DATABASE AdventureWorks2012
* SET ALLOW\_SNAPSHOT\_ISOLATION ON;

A transaction running under snapshot isolation can access tables in the database that have been enabled for snapshot. To access tables that have not been enabled for snapshot, the isolation level must be changed. For example, the following code example shows a SELECT statement that joins two tables while running under a snapshot transaction. One table belongs to a database in which snapshot isolation is not enabled. When the SELECT statement runs under snapshot isolation, it fails to execute successfully.

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT t1.col5, t2.col5

FROM Table1 as t1

INNER JOIN SecondDB.dbo.Table2 as t2

ON t1.col1 = t2.col2;

The following code example shows the same SELECT statement that has been modified to change the transaction isolation level to read-committed. Because of this change, the SELECT statement executes successfully.

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT t1.col5, t2.col5

FROM Table1 as t1

WITH (READCOMMITTED)

INNER JOIN SecondDB.dbo.Table2 as t2

ON t1.col1 = t2.col2;

#### Limitations of Transactions Using Row Versioning-based Isolation Levels

Consider the following limitations when working with row versioning-based isolation levels:

* READ\_COMMITTED\_SNAPSHOT cannot be enabled in tempdb, msdb, or master.
* Global temp tables are stored in tempdb. When accessing global temp tables inside a snapshot transaction, one of the following must happen:
  + Set the ALLOW\_SNAPSHOT\_ISOLATION database option ON in tempdb.
  + Use an isolation hint to change the isolation level for the statement.
* Snapshot transactions fail when:
  + A database is made read-only after the snapshot transaction starts, but before the snapshot transaction accesses the database.
  + If accessing objects from multiple databases, a database state was changed in such a way that database recovery occurred after a snapshot transaction starts, but before the snapshot transaction accesses the database. For example: the database was set to OFFLINE and then to ONLINE, database autoclose and open, or database detach and attach.
* Distributed transactions, including queries in distributed partitioned databases, are not supported under snapshot isolation.
* SQL Server does not keep multiple versions of system metadata. Data definition language (DDL) statements on tables and other database objects (indexes, views, data types, stored procedures, and common language runtime functions) change metadata. If a DDL statement modifies an object, any concurrent reference to the object under snapshot isolation causes the snapshot transaction to fail. Read-committed transactions do not have this limitation when the READ\_COMMITTED\_SNAPSHOT database option is ON.

For example, a database administrator executes the following ALTER INDEX statement.

USE AdventureWorks2012;

GO

ALTER INDEX AK\_Employee\_LoginID

ON HumanResources.Employee REBUILD;

GO

Any snapshot transaction that is active when the ALTER INDEX statement is executed receives an error if it attempts to reference the HumanResources.Employee table after the ALTER INDEX statement is executed. Read-committed transactions using row versioning are not affected.

|  |
| --- |
| **NoteNote** |
| BULK INSERT operations may cause changes to target table metadata (for example, when disabling constraint checks). When this happens, concurrent snapshot isolation transactions accessing bulk inserted tables fail. |

Arrow icon used with Back to Top link[[Top]](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Top)

## Customizing Locking and Row Versioning

### Customizing the Lock Time-Out

When an instance of the Microsoft SQL Server Database Engine cannot grant a lock to a transaction because another transaction already owns a conflicting lock on the resource, the first transaction becomes blocked waiting for the existing lock to be released. By default, there is no mandatory time-out period and no way to test whether a resource is locked before locking it, except to attempt to access the data (and potentially get blocked indefinitely).

|  |
| --- |
| **NoteNote** |
| In SQL Server, use the **sys.dm\_os\_waiting\_tasks** dynamic management view to determine whether a process is being blocked and who is blocking it. In earlier versions of SQL Server, use the **sp\_who** system stored procedure. |

The LOCK\_TIMEOUT setting allows an application to set a maximum time that a statement waits on a blocked resource. When a statement has waited longer than the LOCK\_TIMEOUT setting, the blocked statement is canceled automatically, and error message 1222 (Lock request time-out period exceeded) is returned to the application. Any transaction containing the statement, however, is not rolled back or canceled by SQL Server. Therefore, the application must have an error handler that can trap error message 1222. If an application does not trap the error, the application can proceed unaware that an individual statement within a transaction has been canceled, and errors can occur because statements later in the transaction might depend on the statement that was never executed.

Implementing an error handler that traps error message 1222 allows an application to handle the time-out situation and take remedial action, such as: automatically resubmitting the statement that was blocked or rolling back the entire transaction.

To determine the current LOCK\_TIMEOUT setting, execute the @@LOCK\_TIMEOUT function:

SELECT @@lock\_timeout;

GO

### Customizing Transaction Isolation Level

READ COMMITTED is the default isolation level for the Microsoft SQL Server Database Engine. If an application must operate at a different isolation level, it can use the following methods to set the isolation level:

* Run the [SET TRANSACTION ISOLATION LEVEL](https://technet.microsoft.com/en-us/library/ms173763(v=sql.110).aspx) statement.
* ADO.NET applications that use the **System.Data.SqlClient** managed namespace can specify an IsolationLevel option by using the **SqlConnection.BeginTransaction** method.
* Applications that use ADO can set the Autocommit Isolation Levels property.
* When starting a transaction, applications using OLE DB can call **ITransactionLocal::StartTransaction** with isoLevel set to the desired transaction isolation level. When specifying the isolation level in autocommit mode, applications that use OLE DB can set the DBPROPSET\_SESSION property DBPROP\_SESS\_AUTOCOMMITISOLEVELS to the desired transaction isolation level.
* Applications that use ODBC can set the SQL\_COPT\_SS\_TXN\_ISOLATION attribute by using **SQLSetConnectAttr**.

When the isolation level is specified, the locking behavior for all queries and data manipulation language (DML) statements in the SQL Server session operates at that isolation level. The isolation level remains in effect until the session terminates or until the isolation level is set to another level.

The following example sets the SERIALIZABLE isolation level:

USE AdventureWorks2012;

GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

GO

BEGIN TRANSACTION;

SELECT BusinessEntityID

FROM HumanResources.Employee;

GO

The isolation level can be overridden for individual query or DML statements, if necessary, by specifying a table-level hint. Specifying a table-level hint does not affect other statements in the session. We recommend that table-level hints be used to change the default behavior only when absolutely necessary.

The Database Engine might have to acquire locks when reading metadata even when the isolation level is set to a level where share locks are not requested when reading data. For example, a transaction running at the read-uncommitted isolation level does not acquire share locks when reading data, but might sometime request locks when reading a system catalog view. This means it is possible for a read uncommitted transaction to cause blocking when querying a table when a concurrent transaction is modifying the metadata of that table.

To determine the transaction isolation level currently set, use the DBCC USEROPTIONS statement as shown in the following example. The result set may vary from the result set on your system.

USE AdventureWorks2012;

GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

GO

DBCC USEROPTIONS;

GO

Here is the result set.

Set Option Value

---------------------------- -------------------------------------------

textsize 2147483647

language us\_english

dateformat mdy

datefirst 7

... ...

Isolation level repeatable read

(14 row(s) affected)

DBCC execution completed. If DBCC printed error messages, contact your system administrator.

### Locking Hints

Locking hints can be specified for individual table references in the SELECT, INSERT, UPDATE, and DELETE statements. The hints specify the type of locking or row versioning the instance of the SQL Server Database Engine uses for the table data. Table-level locking hints can be used when a finer control of the types of locks acquired on an object is required. These locking hints override the current transaction isolation level for the session.

For more information about the specific locking hints and their behaviors, see [Table Hints (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms187373(v=sql.110).aspx).

|  |
| --- |
| **NoteNote** |
| The Database Engine query optimizer almost always chooses the correct locking level. We recommend that table-level locking hints be used to change the default locking behavior only when necessary. Disallowing a locking level can adversely affect concurrency. |

The Database Engine might have to acquire locks when reading metadata, even when processing a select with a locking hint that prevents requests for share locks when reading data. For example, a SELECT using the NOLOCK hint does not acquire share locks when reading data, but might sometime request locks when reading a system catalog view. This means it is possible for a SELECT statement using NOLOCK to be blocked.

As shown in the following example, if the transaction isolation level is set to SERIALIZABLE, and the table-level locking hint NOLOCK is used with the SELECT statement, key-range locks typically used to maintain serializable transactions are not taken.

USE AdventureWorks2012;

GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

GO

BEGIN TRANSACTION;

GO

SELECT JobTitle

FROM HumanResources.Employee WITH (NOLOCK);

GO

-- Get information about the locks held by

-- the transaction.

SELECT

resource\_type,

resource\_subtype,

request\_mode

FROM sys.dm\_tran\_locks

WHERE request\_session\_id = @@spid;

-- End the transaction.

ROLLBACK;

GO

The only lock taken that references HumanResources.Employee is a schema stability (Sch-S) lock. In this case, serializability is no longer guaranteed.

In SQL Server 2012, the LOCK\_ESCALATION option of ALTER TABLE can disfavor table locks, and enable HoBT locks on partitioned tables. This option is not a locking hint, but can but used to reduce lock escalation. For more information, see [ALTER TABLE (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms190273(v=sql.110).aspx).

### Customizing Locking for an Index

The SQL Server Database Engine uses a dynamic locking strategy that automatically chooses the best locking granularity for queries in most cases. We recommend that you do not override the default locking levels, which have page and row locking on, unless table or index access patterns are well understood and consistent, and there is a resource contention problem to solve. Overriding a locking level can significantly impede concurrent access to a table or index. For example, specifying only table-level locks on a large table that users access heavily can cause bottlenecks because users must wait for the table-level lock to be released before accessing the table.

There are a few cases where disallowing page or row locking can be beneficial, if the access patterns are well understood and consistent. For example, a database application uses a lookup table that is updated weekly in a batch process. Concurrent readers access the table with a shared (S) lock and the weekly batch update accesses the table with an exclusive (X) lock. Turning off page and row locking on the table reduces the locking overhead throughout the week by allowing readers to concurrently access the table through shared table locks. When the batch job runs, it can complete the update efficiently because it obtains an exclusive table lock.

Turning off page and row locking might or might not be acceptable because the weekly batch update will block the concurrent readers from accessing the table while the update runs. If the batch job only changes a few rows or pages, you can change the locking level to allow row or page level locking, which will enable other sessions to read from the table without blocking. If the batch job has a large number of updates, obtaining an exclusive lock on the table may be the best way to ensure the batch job finishes efficiently.

Occasionally a deadlock occurs when two concurrent operations acquire row locks on the same table and then block because they both need to lock the page. Disallowing row locks forces one of the operations to wait, avoiding the deadlock.

The granularity of locking used on an index can be set using the CREATE INDEX and ALTER INDEX statements. The lock settings apply to both the index pages and the table pages. In addition, the CREATE TABLE and ALTER TABLE statements can be used to set locking granularity on PRIMARY KEY and UNIQUE constraints. For backwards compatibility, the **sp\_indexoption** system stored procedure can also set the granularity. To display the current locking option for a given index, use the INDEXPROPERTY function. Page-level locks, row-level locks, or a combination of page-level and row-level locks can be disallowed for a given index.

|  |  |
| --- | --- |
| **Disallowed locks** | **Index accessed by** |
| Page level | Row-level and table-level locks |
| Row level | Page-level and table-level locks |
| Page level and row level | Table-level locks |

Arrow icon used with Back to Top link[[Top]](https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx#Top)

## Advanced Transaction Information

### Nesting Transactions

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

The following example shows the intended use of nested transactions. The procedure TransProc enforces its transaction regardless of the transaction mode of any process that executes it. If TransProc is called when a transaction is active, the nested transaction in TransProc is largely ignored, and its INSERT statements are committed or rolled back based on the final action taken for the outer transaction. If TransProc is executed by a process that does not have an outstanding transaction, the COMMIT TRANSACTION at the end of the procedure effectively commits the INSERT statements.

SET QUOTED\_IDENTIFIER OFF;

GO

SET NOCOUNT OFF;

GO

CREATE TABLE TestTrans(Cola INT PRIMARY KEY,

Colb CHAR(3) NOT NULL);

GO

CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3) AS

BEGIN TRANSACTION InProc

INSERT INTO TestTrans VALUES (@PriKey, @CharCol)

INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)

COMMIT TRANSACTION InProc;

GO

/\* Start a transaction and execute TransProc. \*/

BEGIN TRANSACTION OutOfProc;

GO

EXEC TransProc 1, 'aaa';

GO

/\* Roll back the outer transaction, this will

roll back TransProc's nested transaction. \*/

ROLLBACK TRANSACTION OutOfProc;

GO

EXECUTE TransProc 3,'bbb';

GO

/\* The following SELECT statement shows only rows 3 and 4 are

still in the table. This indicates that the commit

of the inner transaction from the first EXECUTE statement of

TransProc was overridden by the subsequent rollback. \*/

SELECT \* FROM TestTrans;

GO

Committing inner transactions is ignored by the SQL Server Database Engine. The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction. If the outer transaction is committed, the inner nested transactions are also committed. If the outer transaction is rolled back, then all inner transactions are also rolled back, regardless of whether or not the inner transactions were individually committed.

Each call to COMMIT TRANSACTION or COMMIT WORK applies to the last executed BEGIN TRANSACTION. If the BEGIN TRANSACTION statements are nested, then a COMMIT statement applies only to the last nested transaction, which is the innermost transaction. Even if a COMMIT TRANSACTION transaction\_name statement within a nested transaction refers to the transaction name of the outer transaction, the commit applies only to the innermost transaction.

It is not legal for the transaction\_name parameter of a ROLLBACK TRANSACTION statement to refer to the inner transactions of a set of named nested transactions. transaction\_name can refer only to the transaction name of the outermost transaction. If a ROLLBACK TRANSACTION transaction\_name statement using the name of the outer transaction is executed at any level of a set of nested transactions, all of the nested transactions are rolled back. If a ROLLBACK WORK or ROLLBACK TRANSACTION statement without a transaction\_name parameter is executed at any level of a set of nested transaction, it rolls back all of the nested transactions, including the outermost transaction.

The @@TRANCOUNT function records the current transaction nesting level. Each BEGIN TRANSACTION statement increments @@TRANCOUNT by one. Each COMMIT TRANSACTION or COMMIT WORK statement decrements @@TRANCOUNT by one. A ROLLBACK WORK or a ROLLBACK TRANSACTION statement that does not have a transaction name rolls back all nested transactions and decrements @@TRANCOUNT to 0. A ROLLBACK TRANSACTION that uses the transaction name of the outermost transaction in a set of nested transactions rolls back all of the nested transactions and decrements @@TRANCOUNT to 0. When you are unsure if you are already in a transaction, SELECT @@TRANCOUNT to determine if it is 1 or more. If @@TRANCOUNT is 0, you are not in a transaction.

### Using Bound Sessions

Bound sessions ease the coordination of actions across multiple sessions on the same server. Bound sessions allow two or more sessions to share the same transaction and locks, and can work on the same data without lock conflicts. Bound sessions can be created from multiple sessions within the same application or from multiple applications with separate sessions.

To participate in a bound session, a session calls **sp\_getbindtoken** or **srv\_getbindtoken** (through Open Data Services) to get a bind token. A bind token is a character string that uniquely identifies each bound transaction. The bind token is then sent to the other sessions to be bound with the current session. The other sessions bind to the transaction by calling **sp\_bindsession**, using the bind token received from the first session.

|  |
| --- |
| **NoteNote** |
| A session must have an active user transaction in order for **sp\_getbindtoken** or **srv\_getbindtoken** to succeed. |

Bind tokens must be transmitted from the application code that makes the first session to the application code that subsequently binds their sessions to the first session. There is no Transact-SQL statement or API function that an application can use to get the bind token for a transaction started by another process. Some of the methods that can be used to transmit a bind token include the following:

* If the sessions are all initiated from the same application process, bind tokens can be stored in global memory or passed into functions as a parameter.
* If the sessions are made from separate application processes, bind tokens can be transmitted using interprocess communication (IPC), such as a remote procedure call (RPC) or dynamic data exchange (DDE).
* Bind tokens can be stored in a table in an instance of the SQL Server Database Engine that can be read by processes wanting to bind to the first session.

Only one session in a set of bound sessions can be active at any time. If one session is executing a statement on the instance or has results pending from the instance, no other session bound to it can access the instance until the current session finishes processing or cancels the current statement. If the instance is busy processing a statement from another of the bound sessions, an error occurs indicating that the transaction space is in use and the session should retry later.

When you bind sessions, each session retains its isolation level setting. Using SET TRANSACTION ISOLATION LEVEL to change the isolation level setting of one session does not affect the setting of any other session bound to it.

#### Types of Bound Sessions

The two types of bound sessions are local and distributed.

* Local bound session

Allows bound sessions to share the transaction space of a single transaction in a single instance of the Database Engine.

* Distributed bound session

Allows bound sessions to share the same transaction across two or more instances until the entire transaction is either committed or rolled back by using Microsoft Distributed Transaction Coordinator (MS DTC).

Distributed bound sessions are not identified by a character string bind token; they are identified by distributed transaction identification numbers. If a bound session is involved in a local transaction and executes an RPC on a remote server with SET REMOTE\_PROC\_TRANSACTIONS ON, the local bound transaction is automatically promoted to a distributed bound transaction by MS DTC and an MS DTC session is started.

#### When to Use Bound Sessions

In earlier versions of SQL Server, bound sessions were primarily used in developing extended stored procedures that must execute Transact-SQL statements on behalf of the process that calls them. Having the calling process pass in a bind token as one parameter of the extended stored procedure allows the procedure to join the transaction space of the calling process, thereby integrating the extended stored procedure with the calling process.

In the SQL Server Database Engine, stored procedures written using CLR are more secure, scalable, and stable than extended stored procedures. CLR-stored procedures use the **SqlContext** object to join the context of the calling session, not **sp\_bindsession**.

Bound sessions can be used to develop three-tier applications in which business logic is incorporated into separate programs that work cooperatively on a single business transaction. These programs must be coded to carefully coordinate their access to a database. Because the two sessions share the same locks, the two programs must not try to modify the same data at the same time. At any point in time, only one session can be doing work as part of the transaction; there can be no parallel execution. The transaction can only be switched between sessions at well-defined yield points, such as when all DML statements have completed and their results have been retrieved.

### Coding Efficient Transactions

It is important to keep transactions as short as possible. When a transaction is started, a database management system (DBMS) must hold many resources until the end of the transaction to protect the atomicity, consistency, isolation, and durability (ACID) properties of the transaction. If data is modified, the modified rows must be protected with exclusive locks that prevent any other transaction from reading the rows, and exclusive locks must be held until the transaction is committed or rolled back. Depending on transaction isolation level settings, SELECT statements may acquire locks that must be held until the transaction is committed or rolled back. Especially in systems with many users, transactions must be kept as short as possible to reduce locking contention for resources between concurrent connections. Long-running, inefficient transactions may not be a problem with small numbers of users, but they are intolerable in a system with thousands of users.

#### Coding Guidelines

These are guidelines for coding efficient transactions:

* Do not require input from users during a transaction.

Get all required input from users before a transaction is started. If additional user input is required during a transaction, roll back the current transaction and restart the transaction after the user input is supplied. Even if users respond immediately, human reaction times are vastly slower than computer speeds. All resources held by the transaction are held for an extremely long time, which has the potential to cause blocking problems. If users do not respond, the transaction remains active, locking critical resources until they respond, which may not happen for several minutes or even hours.

* Do not open a transaction while browsing through data, if at all possible.

Transactions should not be started until all preliminary data analysis has been completed.

* Keep the transaction as short as possible.

After you know the modifications that have to be made, start a transaction, execute the modification statements, and then immediately commit or roll back. Do not open the transaction before it is required.

* To reduce blocking, consider using a row versioning-based isolation level for read-only queries.
* Make intelligent use of lower transaction isolation levels.

Many applications can be readily coded to use a read-committed transaction isolation level. Not all transactions require the serializable transaction isolation level.

* Make intelligent use of lower cursor concurrency options, such as optimistic concurrency options.

In a system with a low probability of concurrent updates, the overhead of dealing with an occasional "somebody else changed your data after you read it" error can be much lower than the overhead of always locking rows as they are read.

* Access the least amount of data possible while in a transaction.

This lessens the number of locked rows, thereby reducing contention between transactions.

#### Avoiding Concurrency and Resource Problems

To prevent concurrency and resource problems, manage implicit transactions carefully. When using implicit transactions, the next Transact-SQL statement after COMMIT or ROLLBACK automatically starts a new transaction. This can cause a new transaction to be opened while the application browses through data, or even when it requires input from the user. After completing the last transaction required to protect data modifications, turn off implicit transactions until a transaction is once again required to protect data modifications. This process lets the SQL Server Database Engine use autocommit mode while the application is browsing data and getting input from the user.

In addition, when the snapshot isolation level is enabled, although a new transaction will not hold locks, a long-running transaction will prevent the old versions from being removed from **tempdb**.

### Managing Long-Running Transactions

A long-running transaction is an active transaction that has not been committed or roll backed the transaction in a timely manner. For example, if the beginning and end of a transaction is controlled by the user, a typical cause of a long-running transaction is a user starting a transaction and then leaving while the transaction waits for a response from the user.

A long running transaction can cause serious problems for a database, as follows:

* If a server instance is shut down after an active transaction has performed many uncommitted modifications, the recovery phase of the subsequent restart can take much longer than the time specified by the **recovery interval** server configuration option or by the ALTER DATABASE… SET TARGET\_RECOVERY\_TIME option. These options control the frequency of active and indirect checkpoints, respectively. For more information about the types of checkpoints, see [Database Checkpoints (SQL Server)](https://technet.microsoft.com/en-us/library/ms189573(v=sql.110).aspx).
* More importantly, although a waiting transaction might generate very little log, it holds up log truncation indefinitely, causing the transaction log to grow and possibly fill up. If the transaction log fills up, the database cannot perform any more updates. For more information, see [Troubleshoot a Full Transaction Log (SQL Server Error 9002)](https://technet.microsoft.com/en-us/library/ms175495(v=sql.110).aspx), and [The Transaction Log (SQL Server)](https://technet.microsoft.com/en-us/library/ms190925(v=sql.110).aspx).

#### Discovering Long-Running Transactions

To look for long-running transactions, use one of the following:

* **sys.dm\_tran\_database\_transactions**

This dynamic management view returns information about transactions at the database level. For a long-running transaction, columns of particular interest include the time of the first log record (**database\_transaction\_begin\_time**), the current state of the transaction (**database\_transaction\_state**), and the log sequence number (LSN) of the begin record in the transaction log (**database\_transaction\_begin\_lsn**).

For more information, see [sys.dm\_tran\_database\_transactions (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms186957(v=sql.110).aspx).

* DBCC OPENTRAN

This statement lets you identify the user ID of the owner of the transaction, so you can potentially track down the source of the transaction for a more orderly termination (committing it rather than rolling it back). For more information, see [DBCC OPENTRAN (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms182792(v=sql.110).aspx).

#### Stopping a Transaction

You may have to use the KILL statement. Use this statement very carefully, however, especially when critical processes are running. For more information, see [KILL (Transact-SQL)](https://technet.microsoft.com/en-us/library/ms173730(v=sql.110).aspx).